by
Ray Duncan

Power Programming

Arithmetic Routines For Your Computer Programs, Part 1

■ Handling arithmetic operations in assembly language requires a lot of care and attention to logic; here are some proven routines to add to your programming library to make it easier.

If you program in C, you rarely need concern yourself with the mechanics of arithmetic in your applications. You simply declare your variables as long or short, signed or unsigned, integer or floating. You can then trust the compiler to translate the arithmetic operators in your source code into the proper machine instructions, sequences of instructions, or calls to library routines. You can even mix data types if you wish (multiplying a floating point number by an integer, for example); the compiler will generate the appropriate code to convert (or ''coerce'') the type of one piece of data to match the other.

In assembly language programming, on the other hand, you can't avoid the issues of computer arithmetic and data typing. You must have a solid grasp of signed and unsigned two's complement arithmetic, the CPU's built-in support for the basic arithmetic operations, and the algorithms by which more complex arithmetic operations can be constructed out of the available machine instructions.

In the next several columns, we'll explore some of these subjects together. We will begin with the Intel 80*x*86's native support for single- and double-precision integer arithmetic, then develop a library of variable precision arithmetic routines, and finally examine the capabilities of the 80*x*87 numeric coprocessor. As usual, the emphasis will be on practical rather than theoretical issues, although I will try to provide some of the more abstract references.

BASIC TERMINOLOGY

There are two pairs of terms that will crop up repeatedly in these discussions of computer arithmetic: single-precision versus double-precision integers, and signed versus unsigned integers. We should agree on the meaning of these terms at the outset.

The maximum size of a single-precision integer varies from machine to machine, but I shall take it always to denote a number that will fit into a general register and that can be operated on conveniently with single machine instructions. It is also a power-of-2 multiple of bytes. On the 8086, 8088, 80286, and the 80386 running in real mode or in 16-bit protected mode, a single-precision integer is 16 bits, or 2 bytes. On the 80386 in 32-bit protected mode, a single-precision integer is 32 bits, or 4 bytes.

As you'd expect, a double-precision integer is twice the size of a single-precision number for a given machine, and again, it is always a power-of-2 multiple of bytes. On the 8086, 8088, 80286, and 80386, in real mode or 16-bit protected mode, a double-precision integer is 32 bits. On the 80386 in 32-bit protected mode, a double-precision integer is 64 bits. Most double-precision integer operations enjoy only primitive support in the 80*x*86 instruction set, and—in the absence of a numeric coprocessor—must be carried out with sequences of machine instructions that are sometimes rather lengthy.

The distinction between signed and unsigned integers is straightforward. In a signed integer, the most significant bit is reserved for the arithmetic sign. The bit is 0 if the number is positive, 1 if the number is negative. The remaining bits indicate the number's magnitude. The range for a 16-bit signed integer, for example, is from –32,768 (FFFFH) to 32,767 (7FFFH). In an unsigned integer, all bits, including the significant bit, indicate magnitude. A 16-bit unsigned integer can range from 0 to 65,535 (FFFFH).

But wait a minute, you may say—that unsigned 65,535 looks just like a signed –32,768! You're quite right: bits are bits, and the ''signedness'' or ''unsignedness'' of a given bit pattern depends strictly on your point of view. But picking the right point of view is very important; a logical error in which a signed integer is treated as unsigned or vice versa can be the cause of quite subtle and difficult program bugs, as we shall see later.

SINGLE-PRECISION INTEGER ARITHMETIC

The 80*x*86 CPU family supports single-precision integer addition, subtraction, multiplication, and division with the following instructions:

ADD	single-precision addition
SUB	single-precision subtraction
MUL	Unsigned single-precision multiplication
IMUL	Signed single-precision multiplication
DIV	Unsigned single-precision division
IDIV	Signed single-precision division

The instructions listed above are your fundamental tools for working with the family of single-precision integers, and you must be thoroughly familiar with their behavior, as well as any of their idiosyncrasies. These related, but less important, instructions are

NEG	Two's complement (multiply by -1)
CMP	Compare single-precision integers
CBW	Sign-extend 8-bits to 16-bits

ADD, SUB, NEG, and CMP set the CPU's flags (sign, carry, overflow, and zero are the most important) according to the result of the operation. Actually, CMP can be thought of as a sort of nondestructive SUB that doesn't do anything but set the CPU flags (this is an easy way to remember the order of CMP's operands).

You probably noted that ADD, SUB, and CMP do not come in "signed" and "unsigned" varieties. This is because the "signed" or "unsigned" nature of the result is solely in the eye of the beholder. If you want to regard the result as unsigned, you test the carry flag; if you prefer to think of the result as signed, you test the sign and overflow flags. The 80*x*86 family has an astonishingly diverse battery of conditional jumps to provide for this and similar contingencies.

For example, if you're performing a conditional branch after comparing two addresses (which are unsigned values), you would use the JB, JBE, JA, or JAE instructions. After comparing two dollar amounts (signed values), you would use the JL, JLE, JG, or JGE instructions. Testing the wrong flags or selecting the wrong type of conditional jump is a common cause of obscure program bugs—particularly when addresses are being calculated or compared. Such bugs may lie dormant for a long time and then bite suddenly when a change is made to a completely unrelated part of the program.

The multiply and divide instructions are a little more interesting and a little less regular. MUL and IMUL affect only the carry and overflow flags, leaving the rest undefined; DIV and IDIV leave the state of all flags undefined. The signed instructions—IMUL and IDIV—have slightly less range because they render special treatment to the sign bit. Obviously, the unsigned instructions—MUL and DIV—should always be used when you are working with addresses.

Earlier, I asserted glibly that the multiply and divide instructions are single-precision arithmetic operations. The whole truth is not so simple. The multiply instructions accept two single-precision operands, but they produce a double-preci-

DMUL.ASM — **COMPLETE LISTING**

```
        title   DMUL.ASM Double Precision Unsigned Multiply
        page    55,132

; DMUL.ASM      Double Precision Unsigned Multiply
;               for 8086, 8088, 80286, and
;               80386 in real mode/16-bit protected mode
;
; Copyright (C) 1989 Ziff Communications Co.
; PC Magazine * Ray Duncan
;
; Call with:    DX:AX           = double-precision argument 1
;               CX:BX           = double-precision argument 2
;
; Returns:      DX:CX:BX:AX     = quad-precision product
;
; Destroys:     nothing

_TEXT   segment word public 'CODE'

w0      equ     word ptr [bp-2]         ; local variables
w1      equ     word ptr [bp-4]
w2      equ     word ptr [bp-6]
w3      equ     word ptr [bp-8]

        assume  cs:_TEXT

        public  dmul
dmul    proc    near

        push    si                      ; save registers
        push    di
        push    bp                      ; set up stack frame
        mov     bp,sp                   ; for forming result
        sub     sp,8

        mov     di,dx                   ; save copy of argument 1
        mov     si,ax

        mul     bx                      ; arg1 low * arg2 low
        mov     w0,ax
        mov     w1,dx

        mov     ax,di                   ; arg1 high * arg2 high
        mul     cx
        mov     w2,ax
        mov     w3,dx

        mov     ax,di                   ; arg1 high * arg2 low
        mul     bx
        add     w1,ax                   ; accumulate result
        adc     w2,dx
        adc     w3,0

        mov     ax,si                   ; arg1 low * arg2 high
        mul     cx
        add     w1,ax                   ; accumulate result
        adc     w2,dx
        adc     w3,0

        pop     dx                      ; load quad-precision result
        pop     cx
        pop     bx
        pop     ax

        pop     bp                      ; restore registers
        pop     di
        pop     si
        ret                             ; and exit

dmul    endp

_TEXT   ends

        end
```

Figure 1: Here is a double-precision assembly language multiplication routine for the 8086, 8088, 80286, and 80386. It accepts two 32-bit arguments and returns a 64-bit result.

sion result. One argument must always be in register AX, while the other can be in any other register or in memory; the result always appears in registers DX and AX, with the most significant part in DX. The conventional notation for this latter situation is DX:AX. (On the 80386 in 32-bit protected mode, EAX and EDX are used instead of AX and DX.)

The divide instructions accept a double-precision dividend and a single-precision divisor, and they produce a single-precision quotient and remainder. The dividend is always taken from DX:AX; the divisor can be in any other register or in memory. The quotient is always left in register AX, while the remainder appears in DX. (Again, on the 80386 in 32-bit protected mode, EAX and EDX are used instead of AX and DX.)

Why this mixing of single and double-precision arguments and results, and why this special treatment of DX and AX? The reason is that you need to be able to use the multiply and divide instructions to scale a single-precision value (by multiplying, then dividing) through a double-precision intermediate without losing any precision. Use of dedicated registers to provide arguments or to accept results is an explicit trade-off of instruction set orthogonality for more compact opcodes and therefore smaller programs.

As an aside, it is interesting to note the claims by Apple (and Motorola) that the 68000 in the original Macintosh is a 32-bit microprocessor. In spite of the fact that the 68000 has 32-bit registers, its multiply instruction works on 16-bit arguments to generate 32-bit results, and its divide instruction returns 16-bit results. This alone is sufficient to reveal the 68000 as what it is: a 16-bit microprocessor that happens to have a lot of address lines! Only in the 68020 and 68030 (used in the Mac SE/30 and various Mac II models) do we find the true 32- by 32-bit multiply and 64- by 32-bit divide that are diagnostic of a true 32-bit processor.

The 80286 and 80386 support an odd—but handy—form of the IMUL instruction that is not found on the 8086 and 8088. It is one of the very few instructions in the entire 80*x*86 family that has three operands: the destination is always a register; one of the source operands is a register or memory address; and the other is an "immediate" or literal value. This form of IMUL has a number of other peculiarities: the result of the operation is a single-precision value rather than double; the result can go to a register other than AX and DX; a register argument need not be in AX or DX; and one of the arguments is not (necessarily) destroyed by the operation. For example, to multiply the contents of CX by 10 and leave the result in register BX, you

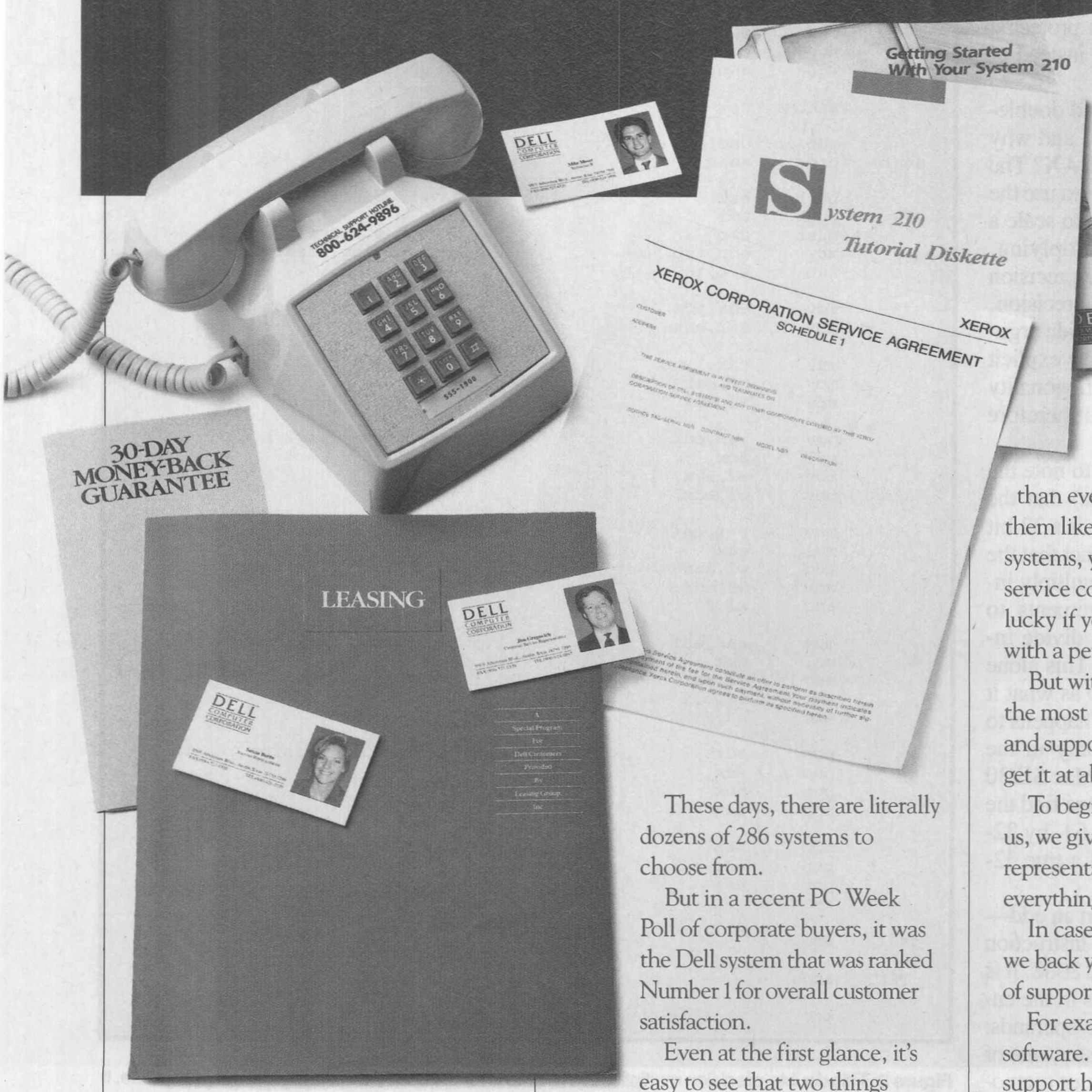
DMUL386.ASM **COMPLETE LISTING**

AVAILABLE ON PC MAGNET

```
        title   DMUL386.ASM Double Precision Unsigned Multiply
        page    55,132
        .386

; DMUL386.ASM   Double Precision Unsigned Multiply
;               for 80386 32-bit protected mode
;
; Copyright (C) 1989 Ziff Communications Co.
; PC Magazine * Ray Duncan
;
; Call with:    EDX:EAX         = double-precision argument 1
;               ECX:EBX         = double-precision argument 2
;
; Returns:      EDX:ECX:EBX:EAX = quad-precision product
;
; Destroys:     nothing

_TEXT   segment dword public use32 'CODE'

w0      equ     dword ptr [ebp-4]       ; local variables
w1      equ     dword ptr [ebp-8]
w2      equ     dword ptr [ebp-12]
w3      equ     dword ptr [ebp-16]

        assume  cs:_TEXT

        public  dmul
dmul    proc    near

        push    esi                     ; save registers
        push    edi
        push    ebp                     ; set up stack frame
        mov     ebp,esp                 ; for forming result
        sub     esp,16

        mov     edi,edx                 ; save copy of argument 1
        mov     esi,eax

        mul     ebx                     ; arg1 low * arg2 low
        mov     w0,eax
        mov     w1,edx

        mov     eax,edi                 ; arg1 high * arg2 high
        mul     ecx
        mov     w2,eax
        mov     w3,edx

        mov     eax,edi                 ; arg1 high * arg2 low
        mul     ebx
        add     w1,eax                  ; accumulate result
        adc     w2,edx
        adc     w3,0

        mov     eax,esi                 ; arg1 low * arg2 high
        mul     ecx
        add     w1,eax                  ; accumulate result
        adc     w2,edx
        adc     w3,0

        pop     edx                     ; load quad-precision result
        pop     ecx
        pop     ebx
        pop     eax

        pop     ebp                     ; restore registers
        pop     edi
        pop     esi
        ret                             ; and exit

dmul    endp

_TEXT   ends

        end
```

Figure 2: This double-precision multiplication routine is for the 80386 CPU in 32-bit protected mode. It accepts two 64-bit arguments and returns a 128-bit result.

FOR $1699, YOU CAN GET CUSTOM CONFIGURATION, SELF-DIAGNOSTIC SOFTWARE, A 30-DAY MONEY-BACK GUARANTEE AND NEXT-DAY DESKSIDE SERVICE.



These days, there are literally dozens of 286 systems to choose from.

But in a recent PC Week Poll of corporate buyers, it was the Dell system that was ranked Number 1 for overall customer satisfaction.

Even at the first glance, it's easy to see that two things make our System 210 different from all the rest.

What we put behind it. And what we put in it. Because we build every one of these systems a little bit differently than everybody else. And we support them like nobody else. With some systems, you can buy an expensive service contract. With others, you're lucky if you get a parent company with a permanent address.

But with our System 210, you get the most complete package of service and support in the industry. And you get it at absolutely no charge.^

To begin with, when you buy from us, we give you a personal account representative. Who gets to know everything about you and your system.

In case you ever have a problem, we back you up with multiple levels of support.

For example, our self-diagnostic software. And our toll-free technical support hotline. Between these two,

would write

```
IMUL     BX,CX,10
```

I should mention that MUL, IMUL, DIV, and IDIV additionally support "half-precision" operations (operating on or returning 8-bit values). These are rarely used in the course of normal application programming and will not be referred to further in these columns.

DOUBLE-PRECISION INTEGERS

The 80*x*86 family's support for double-precision operations is discouragingly meager. In addition to the arithmetic instructions we've already considered, you are provided with only the following:

ADC	Single-precision addition with carry
SBB	Single-precision subtraction with carry (borrow)

These instructions, in essence, allow you to propagate the carry bit through the piecewise addition and subtraction of multiple-precision values. For example, to add a double-precision value in DX:AX to a double-precision value in SI:DI, leaving the result in DX:AX, you would write

```
ADD      AX,DI  ; lower half
ADC      DX,SI  ; upper half
```

Similarly, to subtract a double-precision value in SI:DI from a double-precision value in DX:AX, leaving the result in DX:AX, you would write

```
SUB      AX,DI  ; lower half
SBB      DX,SI  ; upper half
```

Other loosely related instructions, useful mainly for conversion of single-precision values to double-precision, are

CWD	Sign-extend 16-bits to 32-bits
CDQ	Sign-extend 32-bits to 64-bits (80386 only)
MOVSX	Sign-extend 8-bits or 16-bits to 16-bits or 32-bits (80386 only)
MOVZX	Zero-extend 8-bits or 16-bits to 16-bits or 32-bits (80386 only)

To take the two's complement of a double-precision number, you can use the time-tested technique of flipping all the bits and then adding 1. For example, to change the sign of a double-precision number in DX:AX, you would write

```
NOT      DX
NOT      AX
ADD      AX,1
ADC      DX,0
```

A slightly faster technique relies on the fact that NEG sets the carry flag:

```
NEG      DX
NEG      AX
SBB      DX,0
```

What about double-precision multiplication and division? Taking the single-precision native MUL, IMUL, DIV, and IDIV instructions as our guide, we know

DDIV.ASM — **COMPLETE LISTING** — AVAILABLE ON PC MAGNET

```
        title   DDIV.ASM Double Precision Unsigned Divide
        page    55,132

; DDIV.ASM      Double Precision Unsigned Divide
;               for 8086, 8088, 80286, and
;               80386 in real mode/16-bit protected mode
;
; Copyright (C) 1989 Ziff Communications Co.
; PC Magazine * Ray Duncan
;
; Call with:    DX:CX:BX:AX     = quad-precision dividend
;               SI:DI           = double-precision divisor
;
; Returns:      DX:AX           = double-precision quotient
;               CX:BX           = double-precision remainder
;
; Destroys:     SI, DI

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT

        public  ddiv
ddiv    proc    near

        push    bp                      ; save register
        mov     bp,cx                   ; BP = 3sw of dividend
        mov     cx,32                   ; initialize loop counter
        clc                             ; carry flag initially clear

ddiv1:  rcl     ax,1                    ; test this bit of dividend
        rcl     bx,1
        rcl     bp,1
        rcl     dx,1
        jnc     ddiv3                   ; jump if bit was clear

ddiv2:  sub     bp,di                   ; subtract divisor from dividend
        sbb     dx,si
        stc                             ; force carry flag set and
        loop    ddiv1                   ; shift it into forming quotient
        jmp     ddiv5

ddiv3:  cmp     dx,si                   ; dividend > divisor?
        jc      ddiv4                   ; no, jump
        jne     ddiv2                   ; yes, subtract divisor
        cmp     bp,di
        jnc     ddiv2                   ; yes, subtract divisor

ddiv4:  clc                             ; force carry flag clear and
        loop    ddiv1                   ; shift it into forming quotient
ddiv5:  rcl     ax,1                    ; bring last bit into quotient
        rcl     bx,1
        mov     cx,bp
        xchg    dx,bx                   ; put quotient in DX:AX
        xchg    cx,bx                   ; put remainder in CX:BX

        pop     bp                      ; restore register
        ret                             ; and exit

ddiv    endp

_TEXT   ends

        end
```

Figure 3: Corresponding to Figure 1, this double-precision division routine is for the 8086, 8088, 80286, and 80386 (real or 16-bit protected mode). It accepts a 64-bit dividend and 32-bit divisor, returning a 32-bit quotient and 32-bit remainder.

that a truly useful double-precision multiply must process two double-precision arguments to produce a quad-precision result. Similarly, a fully generalized double-precision divide must accept a quad-precision dividend and double-precision divisor, yielding a double-precision quotient and remainder.

At this point, your intuition as a veteran 80*x*86 programmer is probably whispering that you are about to run short of registers. The problems actually go far deeper than this, however. You might reasonably hope that the built-in single-precision multiply and divide instructions could be employed as useful building blocks for double-precision (or multiple-precision) multiply and divide routines. Unfortunately, fate is not so kind.

Knuth describes an algorithm for a multiple-precision divide that is constructed on single-precision divides, but it's quite complex and not very fast.

The hard reality is that the hardware's single-precision multiply instruction is only marginally helpful when used for stepwise multiple-precision multiplication operations in the ''obvious'' manner. That's because MUL and IMUL are quite slow on the older 8086 and 8088 processors. As for multiple-precision divides, the hardware's built-in divide instruction is (for all practical purposes) useless. Although Donald Knuth has described an algorithm for a multiple-precision divide that is constructed on single-precision divides, it is quite complex and—worse yet—not really very fast.

In the next installment I'll discuss the hoary shift-and-add (for multiplication) er history. We'll then use these algorithms as the basis of multiple-precision multiplication and division routines capable of processing arguments of any size. In the meantime—just to tide you over and give you some code to look at—Figures 1, 2, 3, and 4 contain the source code for double-precision multiplication and division subroutines that are somewhat faster (because listings.

THE IN-BOX

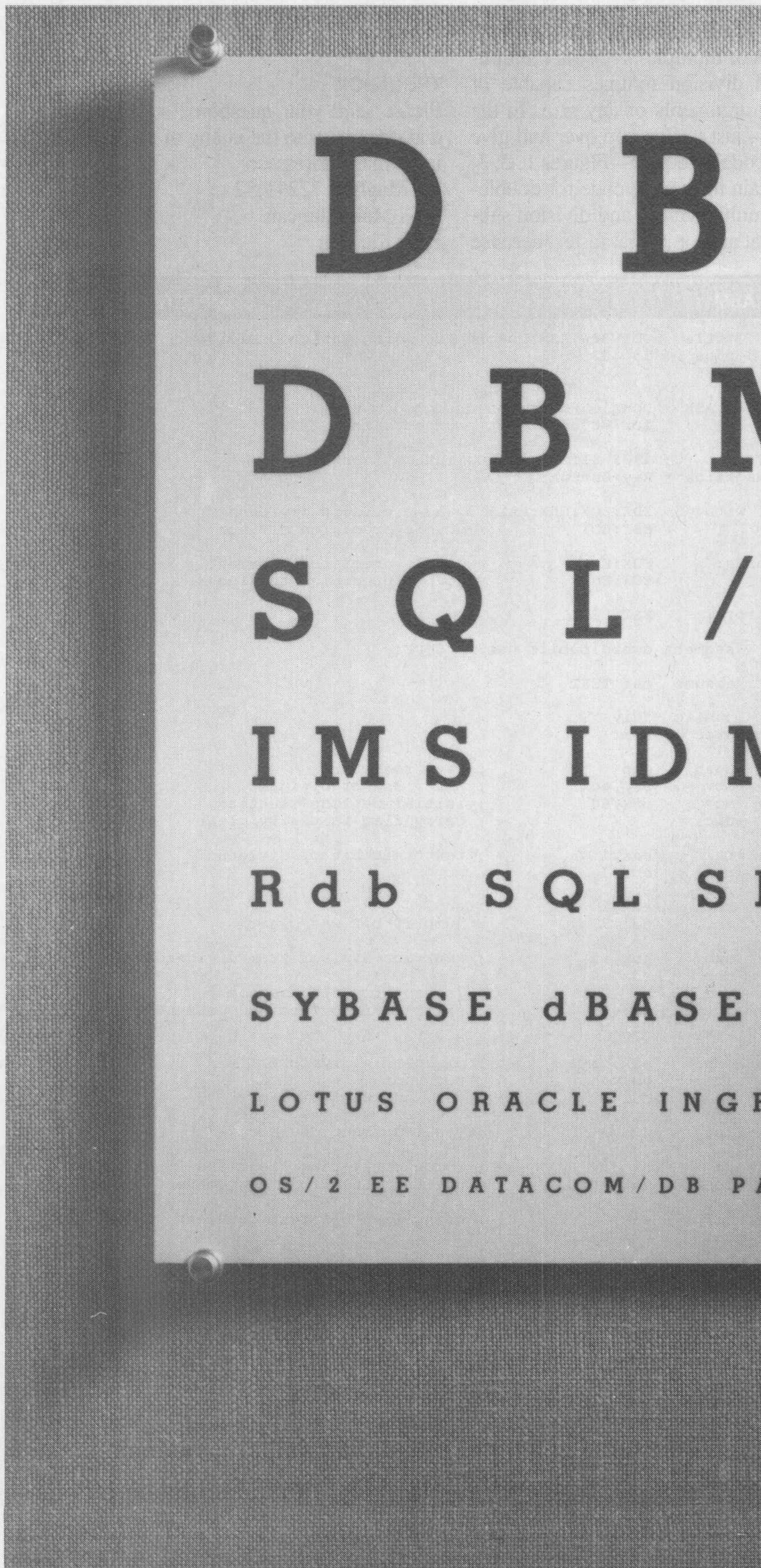
Please send your questions, suggestions, and comments to me at any of the following e-mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan ■

DDIV386.ASM — **COMPLETE LISTING**

AVAILABLE ON PC MAGNET

```
        title   DDIV386.ASM Double Precision Unsigned Divide
        page    55,132
        .386

; DDIV386.ASM   Double Precision Unsigned Divide
;               for 80386 32-bit protected mode
;
; Copyright (C) 1989 Ziff Davis Communications
; PC Magazine * Ray Duncan
;
; Call with:    EDX:ECX:EBX:EAX = quad-precision dividend
;               ESI:EDI         = double-precision divisor
;
; Returns:      EDX:EAX         = double-precision quotient
;               ECX:EBX         = double-precision remainder
;
; Destroys:     ESI, EDI

_TEXT   segment dword public use32 'CODE'

        assume  cs:_TEXT

        public  ddiv
ddiv    proc    near

        push    ebp                     ; save register
        mov     ebp,ecx                 ; EBP = 3sw of dividend
        mov     ecx,64                  ; initialize loop counter
        clc                             ; carry flag initially clear

ddiv1:  rcl     eax,1                   ; test this bit of dividend
        rcl     ebx,1
        rcl     ebp,1
        rcl     edx,1
        jnc     ddiv3                   ; jump if bit was clear

ddiv2:  sub     ebp,edi                 ; subtract divisor from dividend
        sbb     edx,esi
        stc                             ; force carry flag set and
        loop    ddiv1                   ; shift it into forming quotient
        jmp     ddiv5

ddiv3:  cmp     edx,esi                 ; dividend > divisor?
        jc      ddiv4                   ; no, jump
        jne     ddiv2                   ; yes, subtract divisor
        cmp     ebp,edi
        jnc     ddiv2                   ; yes, subtract divisor

ddiv4:  clc                             ; force carry flag clear and
        loop    ddiv1                   ; shift it into forming quotient

ddiv5:  rcl     eax,1                   ; bring last bit into quotient
        rcl     ebx,1

        mov     ecx,ebp
        xchg    edx,ebx                 ; put quotient in EDX:EAX
        xchg    ecx,ebx                 ; put remainder in ECX:EBX

        pop     ebp                     ; restore register
        ret                             ; and exit

ddiv    endp

_TEXT   ends

        end
```

Figure 4: This double-precision division routine for the 80386 in 32-bit protected mode accepts a 128-bit dividend and 64-bit divisor and returns a 64-bit quotient and 64-bit remainder.



You need FOCUS to read them all. PC/FOCUS is the only PC 4GL/DBMS database management system that, through a host of connectivity options, gives DOS, OS/2 and LAN users access to all the data—regardless of the database structure or hardware platform.

PC/FOCUS is also the PC implementation of the world's leading Fourth Generation Language. Capable of universal joins and universal access, FOCUS is the perfect development tool for company-critical cross platform applications and for downsizing or upsizing applications. What's more, it's really easy to use.

If you're beginning to see why you should have PC/FOCUS, call or write Information Builders, Inc.

PC/FOCUS

Information Builders, Inc.

1250 Broadway, New York, NY 10001 (212) 736-4433, Ext. 3700

CIRCLE 244 ON READER SERVICE CARD

by
Ray Duncan

Power Programming

Arithmetic Routines For Your Computer Programs, Part 2

Last time, I discussed single- and double-precision integer arithmetic operations on the 80*x*86 family of processors. In this column, I want to generalize those techniques to cover integer addition, subtraction, and multiplication using any precision you might want or need in your programs. In the next installment, I'll talk a little more about multiplication and then address multiple-precision division.

Before beginning, however, a brief digression into the more-treacherous waters of data formats is in order.

■ This installment covers addition, subtraction, and multiplication, both in generalized C-like form and in full assembly language routines.

BIG-ENDIANS AND LITTLE-ENDIANS

When you program in a high-level language, you generally do not need (or want) to know how the component bytes of an arithmetic value are laid out in memory. If you program in an assembly language, on the other hand, an understanding of binary data formats is absolutely vital. If you are going to load an integer from memory into registers (or vice versa), you clearly need to know which end of the integer is which.

You may be surprised to hear that the world is divided into two hostile camps (wags have dubbed them the "Big-Endians" and the "Little-Endians") over this seemingly innocuous issue. The Big-Endians are committed to a data format that puts the most significant byte of an integer at the lowest memory address, the next most significant byte at the next-higher address, and so on. The Little-Endians, by contrast, are firm believers in a data format in which the least significant byte of the number is placed at the lowest memory address and the most significant byte at the highest memory address occupied by the number.

To make the difference in storage techniques more concrete, consider the 32-bit integer 12345678h, which is composed of four bytes. On a Little-Endian CPU, the four bytes would be layed out in memory thus:

```
78h 56h 34h 12h
```

with 78h occupying the lowest address and 12h the highest. On a Big-Endian CPU, on the other hand, again moving upward from the lowest to the highest address, the four bytes would be arranged in memory as follows:

```
12h 34h 56h 78h
```

Although there are many different examples of CPUs that use each of these data formats, the front lines in this silly little war are manned by the Intel 80*x*86 users on the side of the Little-Endians and by the Motorola 680*x*0 users on the side of the Big-Endians. When a Macintosh programmer meets a PC programmer, I'm often amazed at the intensity of the feelings aroused by this seemingly trivial issue.

Before you get involved in any such heated discussions yourself, just remember that these data formats are only conventions and that equally efficient CPUs can be built using either one. The Little-Endian approach, in which the significance of a byte ascends with its address, seems perfectly logical and intuitive to me, but I'll be the first to admit that hex dumps of memory are far easier to read and interpret on a Big-Endian machine. In any event, I'll be using the Little-Endian format exclusively in the arithmetic routines to be developed in this column, both for the sake of consistency and to make it easier to plug in the use of an 80*x*87 numeric coprocessor later.

Interestingly, the 80486 processor has a new instruction, called BSWAP, whose only purpose is to transform a 32-bit data value in a register from Big-Endian format to a Little-Endian format or back again. In other words, it performs the same function on 32-bit values as the XCHG instruction does on 16-bit values. For example, if you had a 32-bit value in register EAX, the instruction

```
BSWAP EAX
```

would be exactly equivalent to (but much faster than) the sequence

```
XCHG AH,AL
ROL  EAX,16
XCHG AH,AL
```

MULTIPLE-PRECISION ALGORITHMS

Whenever you need to perform addition, subtraction, multiplication, or division to a degree of precision beyond what your CPU's native machine instructions support, you are led directly to the so-called classical algorithms for these operations. The *classical algorithms* are the underpinnings of the stepwise, methodical procedures we all learned in grade school, using paper and pencil, for doing arithmetic on numbers with more than one digit. They are called *classical* because their history extends far back before the dawn of the

computer age. In fact, as Donald Knuth points out, the word "algorithm" was used exclusively in this sense for several centuries, before it acquired its modern, more general meaning.

Figures 1 through 3 contain C-like pseudo-code that demonstrates the classical algorithms for addition, subtraction, and multiplication. (I'm deferring multiple-precision division to the next installment.) The pseudo-code shown is modeled on Knuth's MIX assembly language listings in *The Art of Computer Programming, Volume II: Seminumerical Algorithms* (section 4.3), a definitive work that should always be your ultimate recourse on these and related topics.

In Figures 1 through 3, u[] and v[] are arrays that hold arguments in base b, one digit per array element. The actual physical size of each array element is irrelevant, so long as it is large enough to hold a number of magnitude b−1. The value m represents the maximum number of digits in each argument, and the result is formed in array w[]. The variable k represents the "carry," which is set to the excess when the result of an operation does not fit into a single digit.

To illustrate how these arrays and variables are used, let's consider what happens when we add the very first digits of two multiple-precision numbers together. The value of the first result digit and the resulting carry are found as follows:

```
w[0] = (u[0] + v[0]) mod b
k    = (u[0] + v[0]) / b
```

Subsequent digits (1 through m−1) in the result are found in the same way, except that the previous value of k is included, as follows:

```
w[i] = (u[i] + v[i] + k) mod b
k    = (u[i] + v[i] + k) / b
```

One interesting aspect of these classical algorithms is that they apply equally well to numbers in any base whatever. You can choose to view your arguments and results as bit arrays (base 2), or you can group the bits together and work on octal numbers (base 8) or hexadecimal numbers (base 16); you can even allow the natural byte or word size of the CPU to be an individual "digit."

ADDITION PSEUDO-CODE — **COMPLETE LISTING** — AVAILABLE ON PC MAGNET

```
int m;                        // number of digits
int i;                        // index variable
int b;                        // base
int k;                        // carry

array u[m], v[m];             // holds arguments
array w[m];                   // receives results

k = 0;                        // initialize carry

for(i = 0; i < m; i++);       // add digit by digit
{
    w[i] = (u[i] + v[i] + k) mod b
    k    = (u[i] + v[i] + k) / b
}
```

Figure 1: In this simplified, C-like pseudo-code for multiple-precision addition, both arguments and the result are assumed to be nonnegative. The carry k always takes the value 0 or 1.

SUBTRACTION PSEUDO-CODE — **COMPLETE LISTING** — AVAILABLE ON PC MAGNET

```
int m;                        // number of digits
int i;                        // index variable
int b;                        // base
int k;                        // carry

array u[m], v[m];             // holds arguments
array w[m];                   // receives results

k = 0;                        // initialize carry

for(i = 0; i < m; i++);       // subtract digit by digit
{
    w[i] = (u[i] - v[i] + k) mod b
    k    = (u[i] - v[i] + k) / b
}
```

Figure 2: Multiple-precision subtraction in C-like pseudo-code. In this simplified presentation, both arguments and the result are assumed to be nonnegative, and the argument in array u[] is assumed to be greater than or equal to the argument in v[]. The carry k always takes the value 0 or −1.

MULTIPLICATION PSEUDO-CODE — **COMPLETE LISTING** — AVAILABLE ON PC MAGNET

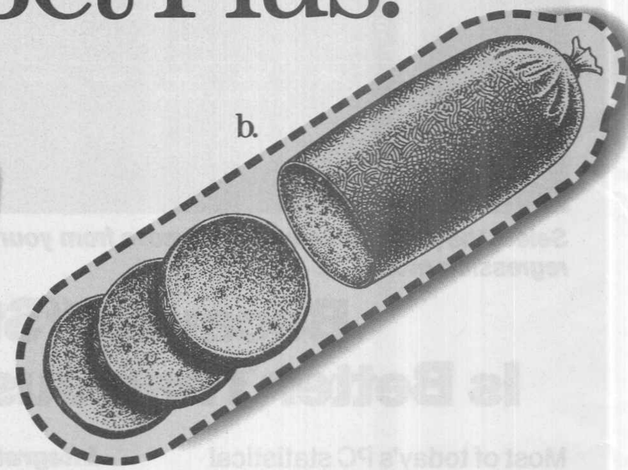
```
int m;                        // number of arg. digits
int i, j;                     // index variables
int b;                        // base
int k;                        // carry
int t;                        // scratch variable

array u[m], v[m];             // holds arguments
array w[m*2];                 // receives product

for(i = 0; i < m*2; i++);     // initialize product
{
    w[i] = 0;
}

for(i = 0; i < m; i++);       // sum partial products
{
    k = 0;                    // initialize carry

    for(j = 0; j < m; j++);   // find this partial product
    {
        t      = u[j] * v[i] + w[i+j] + k;
        w[i+j] = t mod b;     // digit of partial product
        k      = t / b;       // calculate carry
    }

    w[i+m] = k;               // highest digit of
}                             // partial product
```

Figure 3: This C-like pseudo-code for multiple-precision multiplication exploits the CPU's native multiply instruction. The square of the base must be less than or equal to the largest product that can be generated by the hardware's unsigned multiply instruction. In this simplified presentation, both of the arguments and the result are assumed to be nonnegative, and both arguments are the same size. The value of the carry k always satisfies the condition 0 <= k < b, where b is the base.

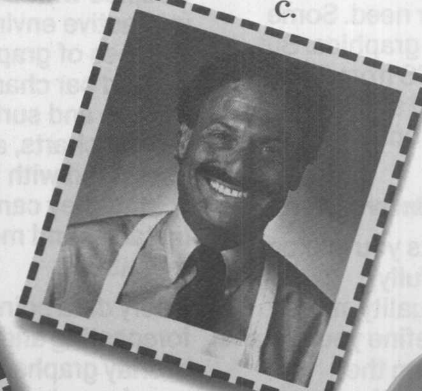
Prove to your boss why you need an HP ScanJet Plus.

a.

b.

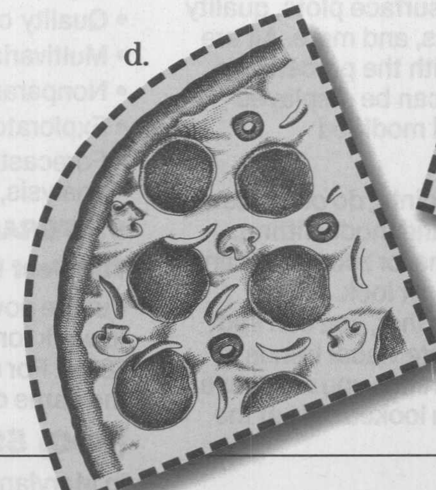


c.



Actual Scanned Images.

d.



1. Cut out the photo and illustrations on the left.

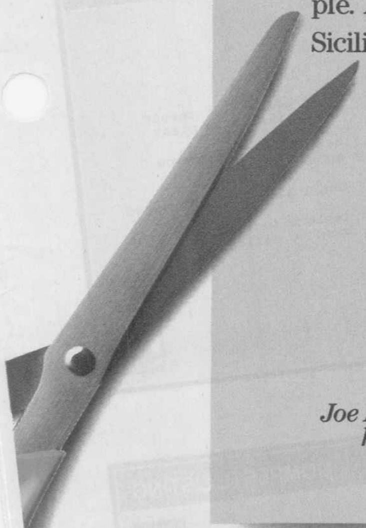
2. Put that dull, wordy document down in front of the boss.

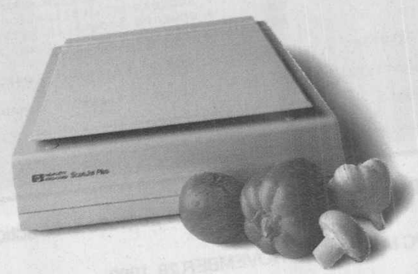
3. For the clincher, lay the graphic elements in place.

Presto! The page comes to life, almost as easily as if you'd scanned the images in with an HP ScanJet Plus scanner. Which you can do for just $2,190.* (In fact, all those images came fresh from our scanner.)

This is a small, but tasty, example of how much you can spice up your communications. With photos, illustrations, and, thanks to OCR, text. Its 8-bit power provides 256 levels of gray and lets you scale images from 4 to 200% in 1% increments.

*U.S. suggested list price. PE12918





The multiplication algorithm shown here differs slightly from the longhand technique you probably learned in school, in that the partial products are accumulated on the fly. When you perform long multiplication with pencil and paper, you normally find all the partial products first, then add them all up at the end of the calculation.

One particularly nice feature of this version of the multiplication algorithm is that it lets you use your CPU's native hardware multiply, if one is available. You need only pick a base such that the square of the base is less than or equal to 1 plus the largest product that can be generated by the CPU's unsigned multiply instruction. If no hardware multiply is available, of course, you simply fall back on base 2, in which case the algorithm degenerates into the

MPNEG.ASM — COMPLETE LISTING

```
        title   MPNEG.ASM Multiple-Precision 2's Complement
        page    55,132

; MPNEG.ASM     Multiple-Precision 2's Complement Routine
;               for Intel 8086, 8088, 80286, and
;               80386 in real mode/16-bit protected mode
;
; Copyright (C) 1989 Ziff Communications Co.
; PC Magazine * Ray Duncan
;
; Call with:    DS:SI   = address of argument
;               CX      = argument length in bytes
;               Assumes direction flag is clear at entry
;
; Returns:      ES:DI   = address of result
;
; Destroys:     Nothing

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT

        public  mpneg
mpneg   proc    near

        mov     di,si                   ; save address of result
        push    cx                      ; save two copies of
        push    cx                      ; argument length

mpneg1: not     byte ptr [si]           ; 1's complement this digit
        inc     si                      ; advance through argument
        loop    mpneg1                  ; until all digits inverted

        pop     cx                      ; retrieve length of argument
        mov     si,di                   ; retrieve first-byte-address
        stc                             ; set carry to add 1

mpneg2: adc     byte ptr [si],0         ; add 1 to 1's complement
        inc     si                      ; to get 2's complement
        loop    mpneg2                  ; until all digits finished

        pop     cx                      ; restore operand length
        mov     si,di                   ; restore argument address
        ret                             ; back to caller

mpneg   endp

_TEXT   ends

        end
```

AVAILABLE ON PC MAGNET

Figure 4: MPNEG.ASM is a general-purpose two's complement routine that changes the sign of multiple-precision integers.

MPADD.ASM — COMPLETE LISTING

```
        title   MPADD.ASM Multiple-Precision Integer Addition
        page    55,132

; MPADD.ASM     Multiple-Precision Integer Addition
;               for Intel 8086, 8088, 80286, and
;               80386 in real mode/16-bit protected mode
;
; Copyright (C) 1989 Ziff Communications Co.
; PC Magazine * Ray Duncan
;
; Call with:    DS:SI   = address of source operand
;               ES:DI   = address of destination operand
;               CX      = operand length in bytes
;               Assumes direction flag is clear at entry
;
; Returns:      ES:DI   = address of result
;
; Destroys:     AL, CX, SI (other registers preserved)

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT

        public  mpadd
mpadd   proc    near

        push    di                      ; save address of result
        clc                             ; carry initially clear

mpadd1: lodsb                           ; next byte from source
        adc     byte ptr es:[di],al     ; accumulate sum
        inc     di
        loop    mpadd1                  ; until all bytes processed

        pop     di                      ; restore address of result
        ret                             ; back to caller

mpadd   endp

_TEXT   ends

        end
```

AVAILABLE ON PC MAGNET

Figure 5: MPADD.ASM is a general-purpose addition routine for multiple-precision integers.

MPSUB.ASM — COMPLETE LISTING

```
        title   MPSUB.ASM Multiple-Precision Integer Subtraction
        page    55,132

; MPSUB.ASM     Multiple-Precision Integer Subtraction
;               for Intel 8086, 8088, 80286, and
;               80386 in real mode/16-bit protected mode
;
; Copyright (C) 1989 Ziff Communications Co.
; PC Magazine * Ray Duncan
;
; Call with:    DS:SI   = address of source operand
;               ES:DI   = address of destination operand
;               CX      = operand length in bytes
;               Assumes direction flag is clear at entry
;
; Returns:      ES:DI   = address of result (destination - source)
;
; Destroys:     AL, CX, SI (other registers preserved)

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT

        public  mpsub
mpsub   proc    near

        push    di                      ; save address of result
        clc                             ; carry initially clear

mpsub1: lodsb                           ; next byte from source
        sbb     byte ptr es:[di],al     ; subtract from destination
        inc     di
        loop    mpsub1                  ; until all bytes processed

        pop     di                      ; restore address of result
        ret                             ; back to caller

mpsub   endp

_TEXT   ends

        end
```

AVAILABLE ON PC MAGNET

Figure 6: MPSUB.ASM is a general-purpose subtraction routine for multiple-precision integers.

MPMUL1.ASM — COMPLETE LISTING

```
        title   MPMUL1.ASM Multiple-Precision Unsigned Multiply
        page    55,132

; MPMUL1.ASM    Multiple-Precision Unsigned Multiply
;               for Intel 8086, 8088, 80286, and
;               80386 in real mode/16-bit protected mode
;
; Copyright (C) 1989 Ziff Davis Communications
; PC Magazine * Ray Duncan
;
; Call with:    DS:SI   = address of source operand
;               ES:DI   = address of destination operand
;               CX      = operand length in bytes
;
;               Assumes direction flag is clear at entry
;               Assumes DS = ES <> SS
;               Assumes CX <= 255
;
; Returns:      ES:DI   = address of product
;
;               NOTE: Buffer for destination operand must be
;               twice as long as the actual operand, because
;               it will receive a double-precision result.
;
; Destroys:     AX (other registers preserved)
;
; Usage:        DS:SI = u[0] base address source operand
;               SS:BP = v[0] base address destination operand
;               ES:DI = w[0] base address of product
;               BX    = i    index for outer loop
;               CX    = j    index for inner loop
;               DH    = m    operand length in bytes
;               DL    = k    remainder of partial products

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT

        public  mpmul1
mpmul1  proc    near

        push    bx                      ; save registers
        push    dx
        push    bp
        sub     sp,cx                   ; make buffer on stack
        mov     bp,sp                   ; for destination operand
        mov     dh,cl                   ; save operand length (m)

        push    cx                      ; copy destination operand
        push    si                      ; to temporary storage in
        push    di                      ; stack frame, because result
        push    es                      ; will be built in destination
        push    ss                      ; operand's buffer
        pop     es
        mov     si,di
        mov     di,bp
        rep     movsb
        pop     es
        pop     di
        pop     si
        pop     cx

        push    di                      ; initialize destination buffer
        xor     ax,ax                   ; to receive result (it better be
        rep     stosw                   ; twice the size of the operands)
        pop     di

        xor     bx,bx                   ; i = 0

mpmul11: xor    dl,dl                   ; k = 0
        xor     cx,cx                   ; j = 0

mpmul12: xchg   bx,cx
        mov     al,[si+bx]              ; get u[j]
        xchg    bx,cx

        xchg    bp,di
        mov     ah,ss:[di+bx]           ; get v[i]
        xchg    bp,di

        mul     ah                      ; t = u[j] * v[i]
        add     al,dl                   ;     + k
        adc     ah,0
        add     bx,cx
        add     al,[bx+di]              ;     + w[i+j]
        adc     ah,0
        mov     [bx+di],al              ; w[i+j] = t mod b
        mov     dl,ah                   ; k      = t / b
        sub     bx,cx                   ; restore i
        inc     cx                      ; j++
        cmp     cl,dh                   ; j = m?
        jne     mpmul12                 ; no, repeat inner loop

        push    bx
        add     bl,dh                   ; w[i+m] = k
        adc     bh,0
        mov     [di+bx],ah
        pop     bx

        inc     bx                      ; i++
        cmp     bl,dh                   ; i = m?
        jne     mpmul11                 ; no, repeat outer loop

        add     sp,bx                   ; discard operand buffer
        pop     bp                      ; restore registers
        pop     dx
        pop     bx
        ret                             ; back to caller

mpmul1  endp

_TEXT   ends

        end
```

Figure 7: MPMUL1.ASM is a general-purpose unsigned multiplication routine for multiple-precision integers.

MPIMUL.ASM — COMPLETE LISTING

```
        title   MPIMUL.ASM Multiple-Precision Signed Multiply
        page    55,132

; MPIMUL.ASM    Multiple-Precision Signed Multiply
;               for Intel 8086, 8088, 80286, and
;               80386 in real mode/16-bit protected mode.
;               Requires MPNEG.ASM (multiple-precision
;               2's complement) and MPMUL1.ASM (multiple-
;               precision unsigned integer multiply).
;
; Copyright (C) 1989 Ziff Davis Communications
; PC Magazine * Ray Duncan
;
; Call with:    DS:SI   = address of source operand
;               ES:DI   = address of destination operand
;               CX      = operand length in bytes
;
;               Assumes direction flag is clear at entry
;               Assumes DS = ES <> SS
;               Assumes CX <= 255
;
; Returns:      ES:DI   = address of product
;
;               NOTE: Buffer for destination operand must be
;               twice as long as the actual operand, because
;               it will receive a double-precision result.
;
; Destroys:     AX (other registers preserved)

_TEXT   segment word public 'CODE'

        extrn   mpmul1:near
        extrn   mpneg:near

        assume  cs:_TEXT

        public  mpimul
mpimul  proc    near

        push    bx                      ; save registers

        mov     bx,cx                   ; take Exclusive-OR of
        mov     al,[si+bx-1]            ; signs of operands
        xor     al,[di+bx-1]
        pushf                           ; save sign of result

        test    byte ptr [si+bx-1],80h  ; source operand negative?
        jz      mpim1                   ; no, jump

        push    di                      ; yes, flip sign of
        call    mpneg                   ; source operand
        pop     di

mpim1:  test    byte ptr [di+bx-1],80h  ; destination operand negative?
        jz      mpim2                   ; no, jump
        push    si                      ; yes, flip sign of
        mov     si,di                   ; destination operand
        call    mpneg
        pop     si

mpim2:  call    mpmul1                  ; perform unsigned multiply

        popf                            ; retrieve sign of result
        jns     mpim3                   ; jump, result is positive

        push    si                      ; operand signs were not
        push    cx                      ; same, make result negative
        mov     si,di
        shl     cx,1
        call    mpneg
        pop     cx
        pop     si

mpim3:  pop     bx                      ; restore register
        ret                             ; back to caller

mpimul  endp

_TEXT   ends

        end
```

Figure 8: MPIMUL.ASM is a general-purpose signed multiplication routine for multiple-precision integers.

more familiar shift-and-add method of multiplication.

MULTIPLE-PRECISION ROUTINES

Figures 4 through 8 are the source listings of the first multiple-precision integer arithmetic modules in our Power Programming library, which are as follows:

`MPNEG`	`Change sign of a multiple-precision integer`
`MPADD`	`Multiple-precision integer addition`
`MPSUB`	`Multiple-precision integer subtraction`
`MPMUL1`	`Multiple-precision unsigned integer multiplication`
`MPIMUL`	`Multiple-precision signed integer multiplication`

The logic of the addition, subtraction, and multiplication routines follows the flow of the pseudo-code listings quite closely. The change-sign procedure employs the familiar trick of taking the one's complement of the entire integer, then adding 1.

The calling sequence for these various multiple-precision routines is documented in their source-code listings. In general, CX is used to pass the length of the arguments, which are assumed always to be the same size. DS:SI points to one argument and ES:DI points to the other. The DS:SI argument is referred to as the *source* and the ES:DI argument as the *destination*, which preserves symmetry with operand usage in the CPU's native ADD and SUB instructions.

The result of the operation always replaces the destination argument, and the address of the result is returned in ES:DI. One warning is in order: when calling the multiple-precision multiply routines, you must make certain that the buffer that holds the destination argument is twice as large as the argument itself so that it will be able to hold the product of the two arguments.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan ■

Power Programming

by
Ray Duncan

Arithmetic Routines For Your Computer Programs, Part 3

■ **Implementations of the classical algorithms for multiplication and division that you can use in your own programs round out this series on arithmetic operations.**

The classical arithmetic algorithms that underlie the longhand procedures we all use for integer addition, subtraction, multiplication, and division have been well understood for hundreds of years. Indeed, the term *algorithm* originally referred only to the formalized procedures for these arithmetic operations and is actually a corruption of the name of renowned Arab mathematician al-Khwārizmī.

The classical algorithms are important not only to schoolchildren but to programmers and computer designers as well. The algorithms are the foundation for hardware adders, multipliers, and dividers, and for the design of software routines that can carry out arithmetic operations that are not supported in hardware.

Schoolchildren are typically taught the *how* without the *why* when it comes to basic arithmetic. I found it quite enlightening to look closely at the classical algorithms (particularly for multiplication and division) and to realize the extent to which I had been basing these longhand procedures on faith rather than understanding.

The aspiring programmer's ultimate resource on the classical algorithms (and on a mind-boggling assortment of other topics as well) is Donald Knuth's *The Art of Computer Programming*. Knuth combines a gift for clear writing with a depth of mathematical insight and a breadth of knowledge and experience that have few parallels in these days of superspecialized professors. Dr. Knuth needs no favorable reviews from me, of course; although his three volumes (so far) may at first appear intimidating, it is a truism to say that they should be on the bookshelves of all but the most casual programmer.

Knuth's discussion of the classical algorithms appears on pages 229–45 of volume 2, *Seminumerical Algorithms*. Unfortunately, however, his program examples are rendered in MIX, the assembly language of a hypothetical CPU for which simulators exist only in the halls of academe. Accordingly, in the last installment of this column, I presented high-level, radix-independent, pseudo-C translations of Knuth's example routines for addition, subtraction, and multiplication. We then used this pseudo-code as a guide for the implementation of corresponding assembly language subroutines.

I don't plan to take this approach for division, however, because the classical algorithm for radix-independent division is rather complex and subtle. If you recall long division as one of the major sore points of your first few years of grade school—something that caused significantly more mental anguish than addition, subtraction, and multiplication—there is a good reason for it. Long division requires normalizations, groupings, and "trial divides" that do not reduce readily into a simple, easily understood piece of radix-independent pseudo-C code.

Luckily, however, there is a solution that will suffice nicely for the purposes of this column. When working in binary (radix = 2), the classical division algorithm degenerates to a considerably simpler form that we typically see implemented in a shift-and-subtract loop. The multiple trial divides that are often needed for each forward step in the generalized form of the algorithm—not to mention the logic necessary to pick trial divisors intelligently—go away completely in binary. Similarly, when used for binary multiplication, the classical algorithm can be simplified into a short and sweet shift-and-add loop.

"Ah yes," I can almost hear you saying, "the good old shift-and-add and shift-and-subtract methods of multiplication and division." Why—even at this considerable distance—can I almost hear you saying this? Because of all the times I've muttered it to myself, of course! We all are familiar with these types of routines, and we feel instinctively that we understand how they work—or could understand easily if we only bothered to try. We have day-to-day experience with using a left shift for a fast multiply by 2 and a right shift for a fast divide by 2. We've all taken the commonly used multiply-by-10 shortcut that relies on a couple of shifts and an add.

But few of us are actually ever called upon to *write* one of these multiplication or division routines, and in practice they are not quite as "obvious" as we fondly imagine. On the other hand, there is certainly nothing magical about such routines; they turn out to be quite straightforward when given the usual attention to detail.

In this column, I'll provide cookbook methods for writing multiplication and division routines that will serve you well on any reasonable CPU (the nasty CPUs that use 1s'-complement arithmetic or lack a carry flag are better avoided than con-

quered), and then I'll illustrate these methods with working code.

RECIPE FOR SHIFT-AND-ADD MULTIPLY

The following procedure assumes that you are multiplying two arguments (sometimes called the multiplier and multiplicand) that are the same length (in bytes) to obtain a product that is twice the length of either argument. The arguments and the product are further assumed to be unsigned; handling arithmetic signs and checking for zero arguments is best done in a "shell" routine external to the fundamental multiplication procedure. This allows routines that need maximum speed and that have control over their arguments to call the unsigned routine directly, achieving best performance. Lastly, it is assumed that your CPU has a carry flag that is under direct program control, and that it has both right and left shift instructions that work together with the carry flag, allowing you to remove a bit from one byte and insert it in another.

Given these assumptions, the steps in the recipe are as follows:

(1) Initialize the high half of the buffer that will receive the product to 0. (The low half will be discarded by shifting, so its original value is unimportant.)

(2) Initialize the loop counter to 8 times the length of each argument (in bytes); this is the number of binary "digits" (bits) in the multiplier that must be tested.

(3) Clear the carry flag.

(4) Logical right-shift the buffer that contains the forming product by one bit position; the value that is in the carry flag becomes the new most significant bit of the product.

(5) Logical right-shift the buffer that contains the second argument (the multiplier) by one position; the "lost" bit shifted out is saved in the carry flag.

(6) If the carry flag is clear (that is, if the bit shifted out of the multiplier was 0), go to step 8.

(7) If the carry flag is set (that is, if the bit shifted out of the multiplier was 1), add the first argument (the multiplicand) to the high half of the forming product. Any overflow of this addition is saved in the carry flag.

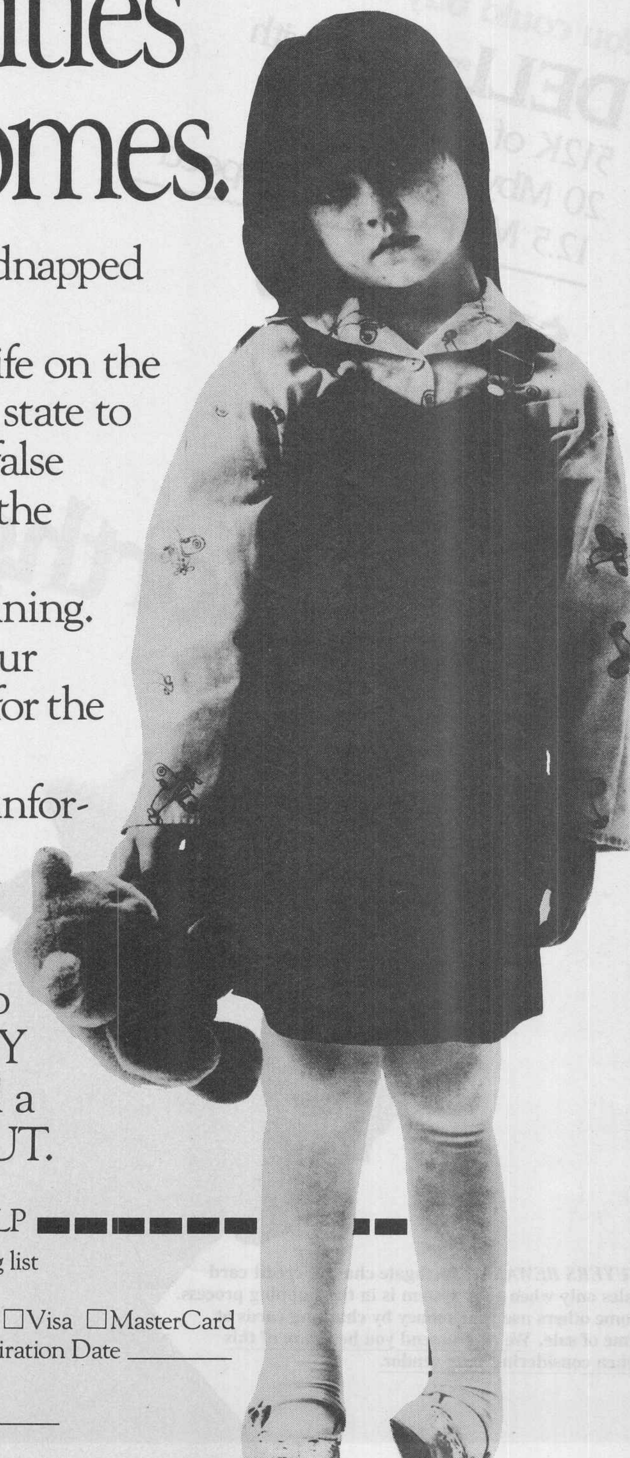
(8) Decrement the loop counter, preserving the carry flag; if the loop counter is nonzero, go to step 4 and continue.

MPMUL2.ASM 1 of 2

```
        title   MPMUL2.ASM Multiple-Precision Unsigned Multiply
        page    55,132

; MPMUL2.ASM    Multiple-Precision Unsigned Multiply
;               for Intel 8086, 8088, 80286, and
;               80386 in real mode/16-bit protected mode.
;               This version uses "shift and add" method.
;
; Copyright (C) 1989 Ziff Communications Co.
; PC Magazine * Ray Duncan
;
; Call with:    DS:SI   = address of source operand
;               ES:DI   = address of destination operand
;               CX      = operand length in bytes
;
;               Assumes direction flag is clear at entry
;               Assumes DS = ES <> SS
;               Assumes 0 < CX <= 255
;
; Returns:      ES:DI   = address of product
;
;               NOTE: Buffer for destination operand must be
;               twice as long as the actual operand, because
;               it will receive a double-precision result.
;
; Destroys:     AX (other registers preserved)
;
_TEXT   segment word public 'CODE'

        assume  cs:_TEXT

        public  mpmul2
mpmul2  proc    near

        push    bx                      ; save registers
        push    cx
        push    dx
        push    bp

        push    di                      ; save addr of dest argument
        mov     dx,cx                   ; save bytes/operand

        add     di,cx                   ; find address of high half
        mov     bp,di                   ; of product, save it in BP

        xor     al,al                   ; initialize high half of
        rep     stosb                   ; forming product to zero

        pop     di                      ; retrieve addr of dest arg

        mov     cx,dx                   ; CX = bits per argument + 1
        shl     cx,1
        shl     cx,1
        shl     cx,1
        inc     cx

        clc                             ; initialize carry

mpmul21: pushf                          ; save carry flag
        mov     bx,dx                   ; BX = bytes in product - 1
        shl     bx,1
        dec     bx
        popf                            ; restore carry flag

mpmul22: rcr    byte ptr es:[di+bx],1   ; shift forming product and
        dec     bx                      ; dest operand right 1 bit
        jns     mpmul22                 ; loop while BX >= 0

        jnc     mpmul24                 ; jump if bit shifted out = 0

                                        ; bit shifted out = 1
        xchg    bp,di                   ; DI = high half of product
        push    cx                      ; save bit counter
        mov     cx,dx                   ; CX = bytes per argument
        xor     bx,bx                   ; init index (also clears carry)

mpmul23: mov    al,[si+bx]              ; add source argument to high
        adc     es:[di+bx],al           ; half of forming product
```

Figure 1: A general-purpose unsigned multiplication routine for multiple-precision integers. This version uses a binary shift-and-add approach that does not exploit the CPU's native hardware multiply. Compare with the MPMUL1.ASM listing published in our previous issue, which carries out multiplication in a byte-wise fashion using the CPU's MUL instruction.

She's 5 years old. She's already had 7 names, 16 identities and 21 homes.



Every year thousands of children are kidnapped by their own parents.

Beginning the lonely ordeal of life on the run. Moving from town to town. From state to state and from school to school. Using false names. And for the parent, living with the constant fear of being discovered.

Child Find offers an end to running. With our special toll-free number and our mediation program, we offer a way out for the parents. And for the kids.

For us, it's a labor of love. But unfortunately, love doesn't pay our operating costs. We need your help. Please give what you can to help Child Find.

Send a check or money order to Child Find, P.O. Box 277, New Paltz, NY 12561-9277. And help another child find a more peaceful future. 1-800-A-WAY-OUT.

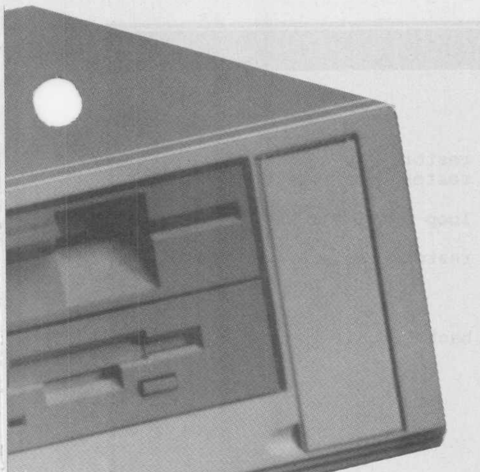
WE NEED YOUR HELP

☐ Please find my gift enclosed ☐ Please put me on your mailing list
☐ Please charge my credit card account
Credit card information (where applicable) ☐American Express ☐Visa ☐MasterCard
Credit Card Number ______________________ Expiration Date ________
Amount you wish to donate ______________
Name ______________________________________
Address ____________________________________
City/State/Zip ______________________________

CHILD FIND OF AMERICA INC.

Mail to Child Find, P.O. Box 277, New Paltz, NY 12561-9277.


You could buy a new
DELL SYSTEM 210 with
512K of RAM
20 Mbyte Hard Drive
12.5 MHz Processor Speed
$1699.00
OR Northgate's™ New 212
NORTHGATE
212
BUYERS BEWARE! Northgate charges credit card sales only when your system is in the shipping process. Some others use your money by charging cards at time of sale. We recommend you be aware of this when considering your vendor.



Microstation with...

1 Full Megabyte RAM
100% GREATER!

32 Mbyte Hard Drive
50% LARGER!

12 MHz Processor Speed

$1699.00

How can Northgate afford to offer a 286/12 with 32MB hard drive and one full megabyte of RAM when Dell's newest Series 210 system, for the same money doesn't match up?

Maybe it's the high cost of color advertising. We use two colors and pass the savings on to our customers. Would you rather get more computer for the money...or be entertained with color pictures?

Here's a fresh idea... Make Northgate and Dell's money-back guarantee program meaningful: Order from both companies.

30 days later send back the loser.

Or keep life simple and place your order with Northgate. Odds are it's the one you'll keep anyway.

NORTHGATE COMPUTER SYSTEMS

"We hear you!"

Northgate Computer Systems, Inc.
P.O. BOX 41000,
Minneapolis, Minnesota 55441

CALL TOLL-FREE 24 HOURS EVERY DAY

800-548-1993

Canada: 800-338-8383

FINANCING: Use the Northgate Big 'N' revolving credit card. We have millions in financing available. We accept your Visa or MasterCard too. Lease it with Northgate, up to five-year terms available.

Prices and specifications are subject to change without notice. Northgate reserves the right to substitute components of equal or greater quality or performance. All items subject to availability.

P.S. Ask about OmniKey/PLUS, the new Northgate Keyboard designed the way you want!

BUYER'S SCORE SHEET.
Read the Specs. Check the Leader.
Total the winning checks and make your decision.

Standard Features	DELL ✓	NORTHGATE ✓
Processor	12.5 MHz 80286	12 MHz 80286
Memory	512K	One Megabyte
Video Interface	16 Bit Built-In (Factory Fixable) Only	16 Bit Add-on (On-Site Fixable)
Display	12" VGA Mono .31 DP	12" VGA Mono .31 DP
Floppy Drive	One - 1.2 or 1.44	One - 1.2 or 1.44
Std. Hard Drive Capacity	20 Mbyte	32 Mbyte RLL
Hard Drive Type	IDE Built-In	Can use - RLL, MFM, IDE, ESDI
I/O Capabilities	2 Ser., 1 PP	2 Ser., 1 PP
Keyboard	"Mushy Touch" 101	Famous OmniKey/102
Software	Diagnostic	On-Line Help, MS-DOS
Space Saver Case	15"W, 15"L, 4"H	14.5"W, 16.5"L, 5.5"H
Moneyback Period	30 Days	30 Days
Warranty	1 Yr. Parts & Labor	1 Yr. Parts & Labor
Phone Tech Support	Unlimited, Toll Free	Unlimited, Toll Free
Hours Open - Sales	Standard Daytime, Eve	24 Hours All Day Every Day
Hours Open - Tech	Standard Daytime	24 Hours All Day Every Day
Total	**$1,699**	**$1,699**
SCORE	**DELL**	**NORTHGATE**

And the winner is...

Call NOW!

To understand what's going on here, just think back to the longhand technique for multiplying decimal numbers. Each digit of the multiplicand is multiplied by each of the digits of the multiplier to obtain a set of partial products. After appropriate shifting, the partial products are added together to form the final product.

In binary multiplication, each "digit" of the multiplier can only be a 0 or a 1, so each "partial product" that needs to be accumulated is either 0 or the appropriately shifted value of the multiplicand. The rest is just trickery to make everything end up in the correct position.

MPMUL2.ASM — 2 of 2

```
            inc     bx
            loop    mpmul23

            pop     cx                          ; restore bit counter
            xchg    bp,di                       ; restore dest operand pointer

mpmul24:    loop    mpmul21                     ; loop until all bits processed

            pop     bp                          ; restore registers
            pop     dx
            pop     cx
            pop     bx
            ret                                 ; back to caller

mpmul2      endp

_TEXT       ends

            end
```

SHIFT-AND-SUBTRACT DIVIDE

In the next procedure, the assumption is that you are dividing an unsigned dividend by an unsigned divisor to get an unsigned quotient and an unsigned remainder. The dividend is further assumed to be twice the length (in bytes) of the divisor; both remainder and quotient are the same length as the divisor.

> **Handling signs, zero divisors, and other odd conditions outside the core unsigned division routine has advantages.**

Again, signs, zero divisors, overflow, and other odd conditions should be handled outside the core unsigned division routine; this allows routines that require maximum speed and that have control over their arguments to call the unsigned routine directly. Finally, it is assumed that the characteristics (shifts and carry-flag control) demanded of the CPU for the shift-and-add multiplication routine also apply for the shift-and-subtract divide routine. The steps in the recipe become:

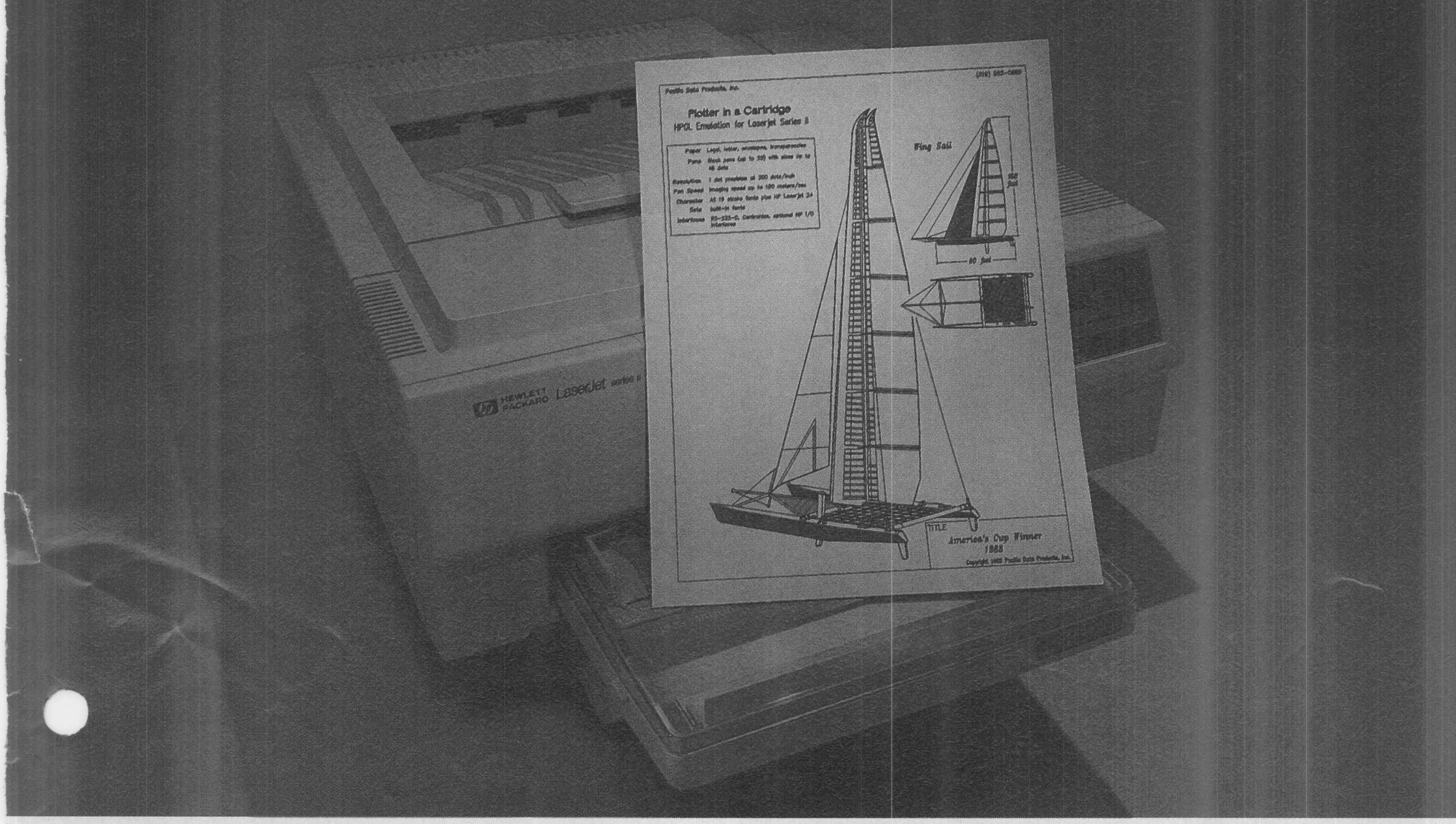
(1) Set the loop counter to the value that is 8 times the length of the divisor (in bytes); this is the number of bits of quotient and remainder that need to be generated. The initial value in the buffer that will receive the

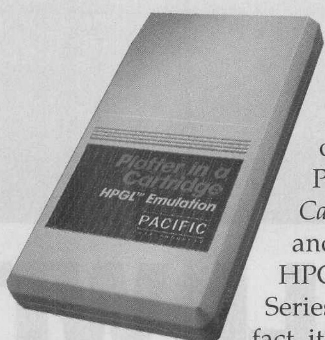
MPDIV.ASM — 1 of 2

```
        title   MPDIV.ASM Multiple-Precision Unsigned Divide
        page    55,132

; MPDIV.ASM     Multiple-Precision Unsigned Divide
;               using "shift-and-subtract" method
;               for Intel 8086, 8088, 80286, and
;               80386 in real mode/16-bit protected mode.
;
; Copyright (C) 1989 Ziff Communications Co.
; PC Magazine * Ray Duncan
;
; Call with:    DS:SI   = address of divisor
;               ES:DI   = address of dividend
;               CX      = divisor length in bytes
;                         (dividend length = 2 * divisor length)
;
;               Assumes direction flag is clear at entry
;               Assumes DS = ES <> SS
;               Assumes 0 < CX <= 255
;
; Returns:      ES:DI   = address of quotient
;               DS:SI   = address of remainder
;
;               NOTE: Dividend is assumed to be twice as long
;               as the divisor.  Returned remainder and quotient
;               are same size as divisor.
;
; Destroys:     AX (other registers preserved)

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT

        public  mpdiv
mpdiv   proc    near

        push    bx                      ; save registers
        push    cx
        push    dx
        push    si
        push    di
        push    bp

        mov     dx,cx                   ; save divisor length in DX

        mov     bp,cx                   ; BP will be outer loop
        shl     bp,1                    ; counter, set it to number
        shl     bp,1                    ; of bits in divisor
        shl     bp,1

        clc                             ; initially clear carry
```

Figure 2: A general-purpose unsigned division routine for multiple-precision integers. It carries out the operation in binary using a shift-and-subtract loop.

The Wait is Over.





HPGL™
Emulation Cartridge

Now you can produce precise, high quality plots without the wait required by pen plotters or PC-based emulations.

Pacific Data's *Plotter in a Cartridge*™ is the newest and fastest way to emulate HPGL on your HP LaserJet Series II or Canon LBP-8II. In fact, it's as much as 100 times faster!

Complex engineering or architectural plots taking 10 to 20 minutes on a plotter can be completed within 10 seconds! Crucial when multiple revisions and check plots are needed.

But there's more to *Plotter in a Cartridge* than speed. In addition to standard pen plotter features, *Plotter in a Cartridge* enables you to define 20 pens and 48 widths, automatically scales to fit envelopes, letter or legal size paper, and improves resolution to 1/300 inch. And, you can produce up to 99 quality copies using virtually any media source.

Because the cartridge plugs right into your printer, you can plot directly from CAD/CAM, engineering, or graphics software.

Pacific Data Products–Plug into Power!

Authorized Dealer:

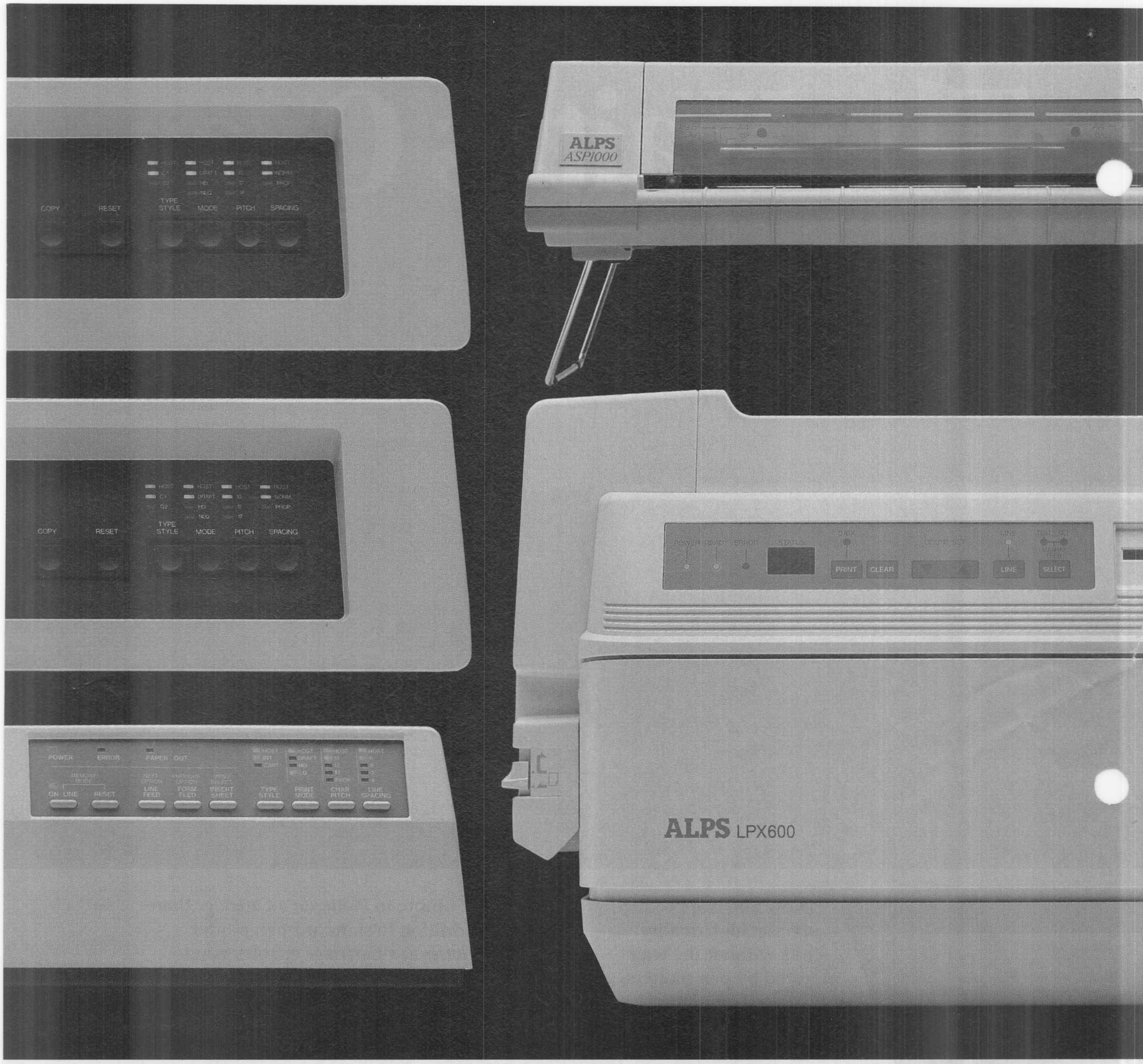
SOFTWARE ETC

800-633-8000

PACIFIC
DATA PRODUCTS

6404 Nancy Ridge Drive, San Diego, CA 92121 • (619) 552-0880 • Fax (619) 552-0889 •

CIRCLE 332 ON READER SERVICE CARD



WE BUILT THEM ALL WIT

We think of it as, well, engineering.

It works like this. You tell us what you need in your printers. We listen. And then our engineers design ALPS printers to do everything you'll ever need printers to do.

For instance, for those who take their printers personally, ALPS offers personal dot matrix printers with jam-proof, flatbed designs and loads of paper handling features. Like our 24-pin Allegro 24, which received the PC MAGAZINE EDITOR'S CHOICE award because, in their words, "The value of the entire package is unmatched."

For the desktop publishers of the world, there's the first in our family of page printers, the compact, lightweight ALPS LPX600. Its performance moved PC WORLD to write, "The print quality is actually superior to the LaserJet Series II's—sharp, crisp type with dense blacks and distinct gray shades."

For everybody else, our workhorse ALQe and "P" Series dot matrix printers can do everything else. From letters to color graphics to high volume financials.

Better yet, every ALPS printer comes with

MPDIV.ASM 2 of 2

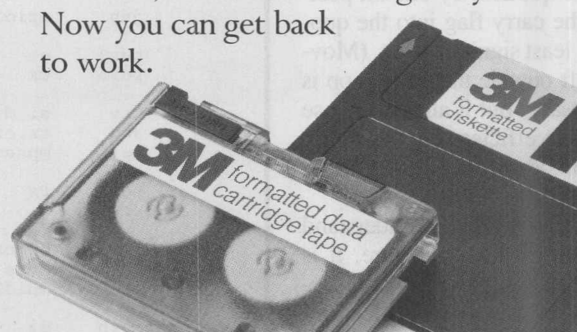
```
mpdiv1: push    di                          ; save pointer to dividend
        mov     cx,dx                       ; CX = bytes in dividend

mpdiv2: rcl     word ptr [di],1             ; shift carry flag into
        inc     di                          ; low bit of quotient
        inc     di                          ; shift high bit of dividend
        loop    mpdiv2                      ; into carry flag

        pop     di                          ; restore pointer to dividend

        jnc     mpdiv5                      ; jump if high bit was clear

mpdiv3: push    si                          ; save pointer to divisor
        push    di                          ; save pointer to dividend

        add     di,dx                       ; DI = addr high half of dividend
        mov     cx,dx                       ; CX = bytes in divisor
        clc                                 ; initially clear carry

mpdiv4: mov     al,[si]                     ; subtract divisor from high
        sbb     [di],al                     ; half of dividend
        inc     si
        inc     di
        loop    mpdiv4

        pop     di                          ; restore pointer to dividend
        pop     si                          ; restore pointer to divisor

        stc                                 ; shift bit=1 into quotient
        dec     bp                          ; all bits of answer generated?
        jnz     mpdiv1                      ; no, loop
        jmp     mpdiv7                      ; yes, go clean up and exit

mpdiv5: push    si                          ; save pointer to divisor
        push    di                          ; save pointer to dividend

        add     di,dx                       ; point to high half of dividend
        mov     cx,dx                       ; CX = bytes in divisor
        clc                                 ; initially clear carry

mpdiv6: mov     al,[di]                     ; high half of dividend > divisor?
        sbb     al,[si]
        inc     si
        inc     di
        loop    mpdiv6

        pop     di                          ; restore pointer to dividend
        pop     si                          ; restore pointer to divisor

        jnc     mpdiv3                      ; jump, high dividend > divisor

        clc                                 ; shift bit=0 into quotient
        dec     bp                          ; all bits of answer generated?
        jnz     mpdiv1                      ; no, loop again

mpdiv7: mov     cx,dx                       ; CX = bytes in quotient

mpdiv8: rcl     byte ptr [di],1             ; bring final bit into quotient
        inc     di
        loop    mpdiv8

        xchg    si,di                       ; copy remainder to final address
        mov     cx,dx
        rep movsb

        pop     bp                          ; restore registers
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        ret                                 ; back to caller

mpdiv   endp

_TEXT   ends

        end
```

Read This And We'll Never Interrupt You Again.

When you're working, there's only one thing worse than being interrupted. Being interrupted by your own PC. But when you stop to format a disk, it makes you stare at a blank screen for about 2 minutes. Format the whole box, and you lose close to 20 minutes. And if you ever have to format a data cartridge tape, your machine is out of action for as much as 2 hours.

But if you use our formatted disks and cartridge tapes, you'll never be interrupted again. The disks come in 3.5″ and 5¼″ sizes. For the IBM PC, PS/2®, and compatibles. Our data cartridge tapes come in 10 to 150 Mbyte versions. All you do is take them out of the box and keep right on working. Of course, it helps to buy them first. So next time you need diskettes or data cartridge tapes, be sure to ask for 3M formatted products.

OK, we've taken enough of your time. Now you can get back to work.



For more information about formatted diskettes and data cartridge tapes, call 1-800-888-1889, ext. 45.
"IBM" and "PS/2" are registered trademarks of the IBM Corporation.

Come see us at Comdex, Booth #1742.

3M

CIRCLE 521 ON READER SERVICE CARD

quotient is unimportant because it will be discarded by shifting during the procedure.
(2) Clear the carry flag.
(3) Left-shift the quotient by one bit position; the previous value of the carry flag is inserted into the quotient as the new least significant bit.
(4) Left-shift the dividend by one bit position; the bit shifted out is saved in the carry flag.
(5) If the carry flag is clear, go to step 7.
(6) Subtract the divisor from the upper half of the dividend. Set the carry flag and go to step 8.
(7) If the upper half of the dividend is larger than the divisor, go to step 6; otherwise, clear the carry flag and go to step 8.
(8) Decrement the loop counter, preserving the state of the carry flag; if the loop counter is nonzero, go to step 3.
(9) Left-shift the quotient by one bit position, bringing the carry flag into the quotient as the final least significant bit. (Moving this last shift outside the main loop is not really necessary, but it allows the use of a slightly more efficient control structure.) The remainder is whatever is left in the high half of the dividend.

Shifting is a shortcut to inspecting groups of the dividend's digits equal in length to the divisor.

Again, when attempting to understand what is going on in this procedure, it is helpful to draw analogies to longhand decimal division. The important distinction, however, is that trial divides are not necessary when we choose to view each bit as a single digit; either the divisor can fit into the portion of the dividend we are looking at or it can't. We use shifting as a convenient shortcut to inspecting groups of the dividend's digits that are the same length as the divisor. The rest is just bookkeeping and positioning of the results.

MPIDIV.ASM 1 of 2

```
        title   MPIDIV.ASM Multiple-Precision Signed Divide
        page    55,132

; MPIDIV.ASM    Multiple-Precision Signed Division
;               for Intel 8086, 8088, 80286, and
;               80386 in real mode/16-bit protected mode.
;               Requires MPNEG.ASM (multiple-precision
;               2's complement) and MPDIV.ASM (multiple-
;               precision unsigned integer divide).
;
; Copyright (C) 1989 Ziff Communications Co.
; PC Magazine * Ray Duncan
;
; Call with:    DS:SI   = address of divisor
;               ES:DI   = address of dividend
;               CX      = divisor length in bytes
;                         (dividend length = 2 * divisor length)
;
;               Assumes direction flag is clear at entry
;               Assumes DS = ES <> SS
;               Assumes 0 < CX <= 255
;
; Returns:      ES:DI   = address of quotient
;               DS:SI   = address of remainder
;
;               NOTE: Dividend is assumed to be twice as long
;               as the divisor.  Returned remainder and quotient
;               are same size as divisor.
;
;               The sign of the quotient is positive if the signs
;               signs of the dividend and divisor are the same;
;               negative if they are different.  The sign of the
;               remainder is the same as the sign of the dividend.
;
; Destroys:     AX (other registers preserved)

_TEXT   segment word public 'CODE'

        extrn   mpdiv:near
        extrn   mpneg:near

        assume  cs:_TEXT

        public  mpidiv
mpidiv  proc    near

        push    bx                      ; save registers

        mov     bx,cx                   ; get Exclusive-OR of
        mov     al,[si+bx-1]            ; signs of operands
        add     bx,bx
        xor     al,[di+bx-1]
        pushf                           ; save sign of result

        mov     al,[di+bx-1]            ; test sign of dividend
        or      al,al
        pushf                           ; save sign of remainder

        jns     mpid1                   ; jump if dividend positive

        push    si                      ; save pointer to divisor
        push    cx                      ; save length of divisor

        mov     si,di                   ; point to dividend
        add     cx,cx                   ; calc length of dividend
        call    mpneg                   ; flip sign of dividend

        pop     cx                      ; restore length of divisor
        pop     si                      ; restore address of divisor

mpid1:  mov     bx,cx                   ; check if divisor negative
        test    byte ptr [si+bx-1],80h
        jz      mpid2                   ; jump, divisor is positive

        push    di                      ; save pointer to dividend
        call    mpneg                   ; flip sign of divisor
        pop     di                      ; restore pointer to dividend

mpid2:  call    mpdiv                   ; perform unsigned divide

        popf                            ; retrieve sign of remainder
        jns     mpid3                   ; jump, remainder is positive
```

Figure 3: A general-purpose signed division routine for multiple-precision integers. This routine requires MPDIV.ASM (Figure 2) and MPNEG.ASM.

Every 500 years or so, an exceptional printer comes along.

Photo credit: Bettmann Archive

Johannes Gutenberg, 1454.

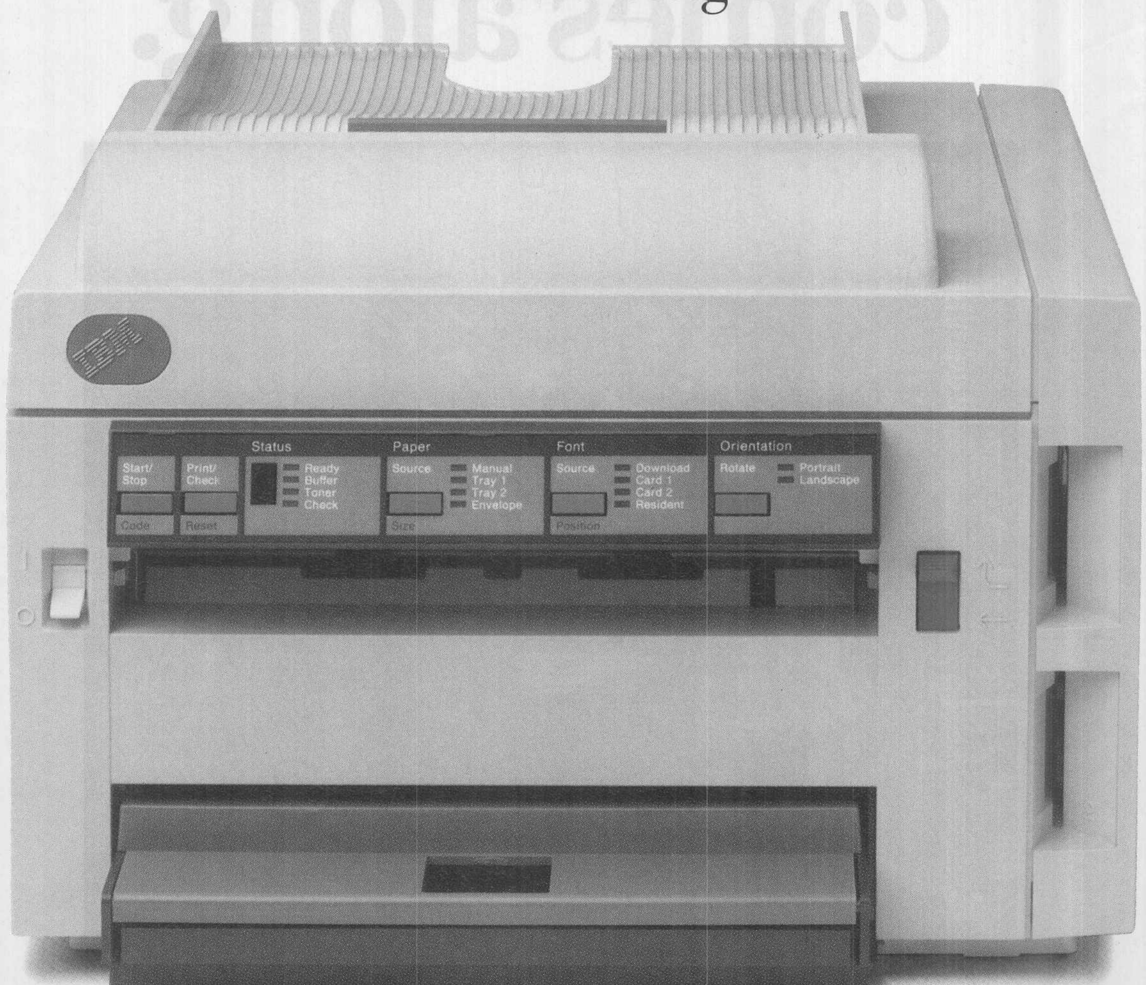
Introducing the new

A printer that can raise your productivity as much as the new IBM LaserPrinter doesn't come along every day.

Not only does its advanced design make it outperform the HP LaserJet Series II, which up till now has been the benchmark in laser printing. But also, its advanced design gives the IBM LaserPrinter a dramatically new, more space-efficient shape.

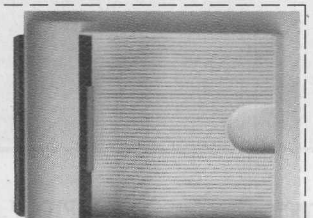
Yet with all this, there's one area in which the competition rises above us: their price. ***We just advanced the art of laser printing 25%.*** The new IBM LaserPrinter gives you state-of-the-art print quality a full 25% faster than its main competitor.

25% faster printing, advanced features and a revolutionary streamlined design.



IBM LaserPrinter.

Why other printers can't follow in our footsteps. The advanced design and engineering of the IBM LaserPrinter give it a footprint that's 33% smaller than its main competitor's. And that 33% gives you more usable workspace.



The space-saving footprint of the IBM LaserPrinter: 33% smaller than its main competitor's.

And more drawer space, too. Because the IBM LaserPrinter's font cards are about the size of credit cards. Yet as compact as they are, they offer twice as many fonts as almost all its main competitors' large, old-fashioned cartridges.

It's only up from there. Since the paper-handling options on the IBM LaserPrinter are stackable, you can add features, like high-capacity paper drawers (500 sheets and 75 envelopes), without sacrificing workspace.

And not only can you connect the IBM LaserPrinter to any IBM and many IBM-compatible PC's, but also, with its printer-sharing option, one IBM LaserPrinter can serve up to 6 PC's, for maximum cost-efficiency.

Another reason it's a new order of laser printer. Unlike its main competitor, the IBM LaserPrinter collates letterheads, second sheets and envelopes.* Most people would rather give that job to someone else anyway. So it's nice to have a printer that will do it for you automatically.

Proof that less is more. The design and engineering of the IBM LaserPrinter are so advanced, it's got 60% fewer parts than the previous standard in laser printing. That means it's got 60% fewer things to loosen, snap or shear.

The solution is IBM. Yet, though it's a smaller, sleeker unit, the IBM LaserPrinter comes from a company with a big tradition of service and support. Which is very reassuring when you decide to break away from an old standard to go with the new one for years to come.

To do that as soon as possible, contact your IBM Authorized Dealer or IBM marketing representative. Find your nearest dealer by calling 1 800 IBM-2468, ext. 194.

		IBM LaserPrinter	HP LaserJet Series II
Speed	✓	up to 10 ppm	up to 8 ppm
Footprint	✓	291 sq. in.	432 sq. in.
Paper-handling options	✓	500 sheets, 75 envelopes	15 envelopes*
Collates letters/envelopes	✓	yes**	no
Plotter emulation	✓	standard	optional
Resident fonts	✓	10	6
Font card size***	✓	credit card	"8-track" cassette
Standard weight	✓	33 lbs.	50 lbs.
Parts***	✓	400	1000
Dots per inch		300 x 300	300 x 300
Printer emulation	✓	IBM, HP compatible	HP compatible
Printer engine		IBM	Canon
List price	✓	$2,595	$2,695

*HP envelope tray replaces standard paper tray **With paper-handling options ***Approximate

The new IBM LaserPrinter.
Suddenly, nothing else measures up.

IBM®

MPIDIV.ASM **2 of 2**

```
                push    di                      ; save pointer to quotient
                call    mpneg                   ; flip sign of remainder
                pop     di                      ; restore pointer to quotient

mpid3:          popf                            ; retrieve sign of result
                jns     mpid4                   ; jump, result is positive

                push    si                      ; save pointer to remainder
                mov     si,di                   ; point to quotient
                call    mpneg                   ; flip sign of quotient
                pop     si                      ; restore pointer to remainder

mpid4:          pop     bx                      ; restore register
                ret                             ; back to caller

mpidiv          endp

_TEXT           ends

                end
```

MULTIPLE-PRECISION ROUTINES

Figures 1 through 3, plus MPIMUL.ASM and MPNEG.ASM presented last time, contain the source code for assembly language procedures that illustrate what we've been discussing here and that round out our battery of multiple-precision arithmetic routines. The calling procedures and results of each routine are documented in the listings.

MPMUL2.ASM, shown in Figure 1, is the unsigned multiple-precision-integer multiplication routine that uses the shift-and-add technique. You may find it instructive to compare this code with the MPMUL1.ASM published here in the previous issue. The latter used the CPU's native 8-bit-by-8-bit multiply, and you may wish to run some timing comparisons of the two routines. When running benchmark tests, remember that there are drastic differences in the cost of a hardware multiply as you progress from the 8086/88 to the 80386 and 80486.

MPIMUL.ASM is the signed multiple-precision multiply routine. It checks the signs of the arguments to determine the sign of the eventual result, changes arguments from negative to positive if necessary (using MPNEG.ASM), then calls MPMUL2.ASM to do the hard work.

MPDIV.ASM, shown in Figure 2, is the unsigned multiple-precision divide routine that implements the shift-and-subtract technique described earlier. If you're feeling spunky, read Knuth (volume 2, pages 237–38) and code a new version of this routine that exploits your CPU's native DIV instruction.

MPIDIV.ASM, shown in Figure 3, is the signed multiple-precision divide routine that checks and changes signs of arguments and results, much in the same way as MPIMUL.ASM. It calls MPNEG.ASM and MPDIV.ASM. Note that calls to MPIDIV.ASM should be avoided if you know that the sign of your arguments and results is not important (for example, when manipulating addresses), since MPIDIV is slower than MPDIV.

I've tried to make these routines reasonably efficient, though to keep them from diverging too far from the recipes presented above, I have forgone a number of optimizations that I would use in a production program. Once you're sure you understand the code, you can entertain yourself for hours by tuning it up further. Just beware of introducing machine instructions that affect the carry flag!

I've also written two interactive demonstration programs, TRYMPMUL.ASM and TRYMPDIV.ASM, that will facilitate your experiments. These programs prompt you for arguments, call the appropriate multiply or divide routine, then display the results. Because of their length, TRYMPMUL and TRYMPDIV are not printed here, but both are available for downloading from PC MagNet.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan

by
Ray Duncan

Power Programming

Arithmetic Routines for Your Computer Programs, Part 4

■ Floating-point numbers present problems not encountered when dealing only with integers, including the question of how to store them in memory.

In the first three columns in this series on computer arithmetic we looked at the classical algorithms for the four basic integer operations. We also devised a set of assembly language subroutines for multiple-precision integer addition, subtraction, multiplication, and division. It's time now to venture forth from the relatively safe shallows of integer arithmetic into the treacherous depths of real numbers and floating-point arithmetic.

A grade school student is introduced to the real numbers in stages. First, he gets the "counting numbers" (positive integers). Next, the negative numbers are added, completing the set of all integers. Finally, he comes to fractions and their decimal equivalents. To make these concepts easier to visualize, the teacher often uses a "number line": a horizontal line with arrows at both ends (pointing to negative infinity on the left and positive infinity on the right) and with a hashmark (signifying zero) in the middle. Once the integers (. . . –3, –2, –1, 0, 1, 2, 3 . . .) are charted on the number line, it's a fairly easy jump to the notion that there are "numbers between the numbers." The fraction ½ is symbolized by putting a dot on the number line halfway between the 0 and the 1, for example. From there it's but another step to the realization that there are an infinite number of numbers between any two arbitrary points on the number line, the whole comprising the set of real numbers.

At some point in high school—if he chooses algebra, chemistry, and physics over machine shop and varsity athletics—our model student is taught "scientific" or "exponential" notation. This is a tool that allows him to write down real numbers of any desired size or precision. For example, the fraction ¼ can be expressed in scientific notation as $2.5*10^{-1}$.

The "2.5" portion of the notation is called the *mantissa* or *fraction* or *significand*. It has one nonzero digit to the left of he decimal point, and the number of digits after the decimal point indicates the degree of precision to which the number's value is known. (A mantissa in this form is said to be "normalized.") The "10^{-1}" portion is called the *exponent* or *characteristic*; it specifies the location of the decimal point in the number. Teachers have a whole cookbook full of rules for operations on numbers written in scientific notation, such as: "To find the product of two numbers, multiply the mantissas and sum the powers of the exponents."

With such mastery of the real numbers and the means to manipulate them in hand, our student, now a regular whiz kid, may be tempted to adopt a somewhat smug attitude toward matters mathematical. Never mind; his prematurely optimistic outlook will be demolished when he's confronted with imaginary numbers, irrational numbers, complex numbers, infinitesimals, and all the other counterintuitive mathematical things that go bump in the night.

FLOATING POINT ON COMPUTERS

While the methods we use to manipulate floating-point numbers on computers are certainly based on the fundamental rules and algorithms we were all taught way back when, there are several important differences we must bear in mind.

For one thing, most computers and high-level language libraries support only a limited number of floating-point formats (typically only two), and these formats, by nature, have a finite precision and range. This means that you cannot possibly represent every real number as a floating-point number on your computer. In fact, the number of numbers that you can't express is infinitely larger than the number of numbers that you can represent. When your CPU or compiler was designed, its creators picked a floating-point data format (or formats) they felt would be sufficient for most "normal" applications and yet could be implemented reasonably efficiently. If the requirements of your particular application program fall outside the bounds foreseen by those designers, you either have to roll your own floating-point routines or do without.

Not only are your computer's floating-point numbers a minuscule subset of the set of real numbers, they do not map onto the real numbers in a uniform way. For example, if you plot the numbers that can be represented by a 32-bit *integer* onto the real number line, you'll see a set of points that march monotonically along the line from –2,147,483,648 (2^{-31}) to 2,147,483,647 (2^{31}–1). But if you now plot the numbers that can be represented by a 32-bit *floating-point* number onto the real number line, you'll be in for a surprise. The number of numbers that can be represented in the floating-point format is exactly the same as for the integer format

(think about it!). But, as illustrated in Figure 1, the floating-point numbers are densely clustered around zero, and become increasingly sparse as the distance from zero increases. Of course, the smallest and largest floating-point numbers are far smaller (or larger) than the smallest and largest integer, but this dynamic range is gained at the expense of precision: there are only so many bits to go around.

Still a third set of new considerations arises from the fact that while people prefer to compute in base 10, computers find base 2 (binary) much more to their liking. As application designers and programmers, we like to keep everybody happy ("from each according to his abilities, to each according to his needs" as the famous coder Karl Marx enjoined in his 1875 tutorial, "Criticism of the Gotha Programme"). To do so, each time a number is input or output it must be converted from decimal to binary or vice versa. This caused no real problems when we were working with integers. It becomes a thorny issue with floating point, however, because some apparently quite ordinary decimal numbers cannot be expressed exactly in binary floating point (one such number is $1.0*10^{-1}$).

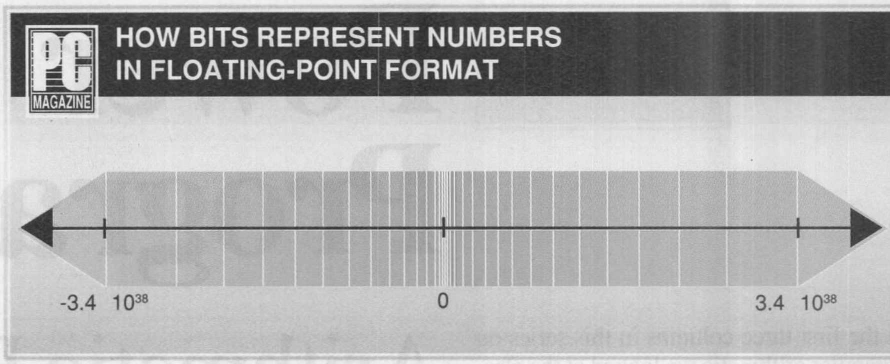


Figure 1: While the number of numbers that can be represented by a given number of bits is the same for integer and floating-point formats, in floating point the *range* of numbers is greater, but they occur at increasing intervals as they get further away from zero.

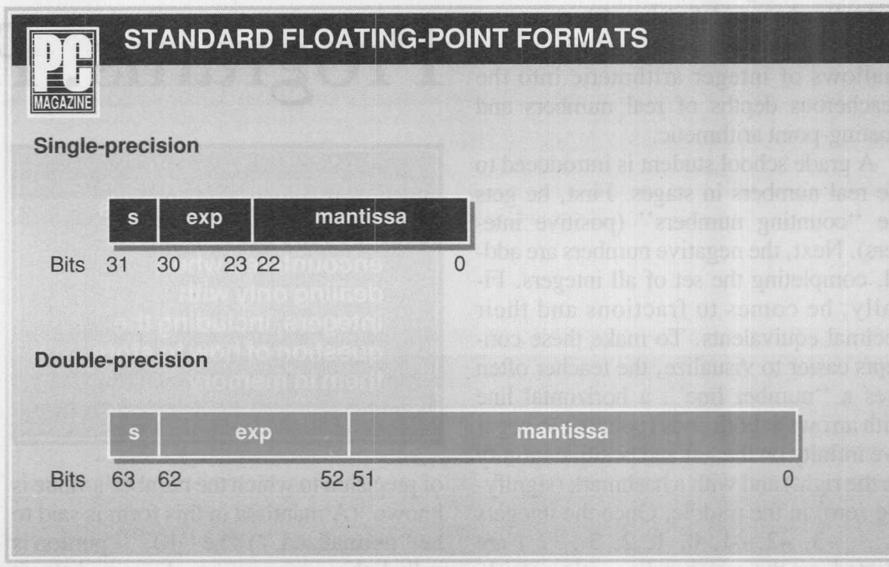


Figure 2: The single-precision and double-precision floating-point formats specified by the ANSI/IEEE 754 Standard for Binary Floating-Point Arithmetic. The Standard also specifies "extended" single and double formats, which are not discussed here.

BINARY FLOATING-POINT DATA FORMATS

Once a decimal floating-point number has been converted to a normalized binary floating number, it can be thought of as having the form

$$1.bbbbb\ldots * 2^{n}$$

Where each bit *b* in the mantissa is a zero or a one. The mantissa is normalized by adjusting the exponent so that the most significant one bit is to the left of the binary point; in other words, the mantissa is always greater than or equal to 1 and less than 2.

How are these floating-point numbers actually stored in memory? The history of this topic is one that begins in utter chaos but (for once) has a happy ending. In the early days of computing, a thousand flowers bloomed and a thousand schools of thought contended, with the inevitable result that nearly every compiler and CPU used a different floating-point data format. This made it very difficult to transport data from one machine to another, or even from a program written in one high-level language to one written in another.

Fortunately, in the late 1970s, a concerted effort to standardize binary floating-point arithmetic was begun, first under the auspices of the ACM, and later under the IEEE Computer Society. This undertaking drew upon several proposals, the most important of which was the so-called KCS Proposal (written by Kahan, Connen, and Stone in 1978). These proposals, in turn, represented an integration of concepts and techniques that dated back to the earliest days of computer science. The IEEE committee's work resulted in the publication in 1981 of the draft IEEE 754 Standard for Binary Floating-Point Arithmetic. This was adopted (in a slightly modified form) as an official ANSI/IEEE Standard in 1985.

The IEEE 754 Standard was principally directed at making floating-point calculations safe and predictable for programmers untrained in numeric analysis (that's nearly all of us!). It did this by specifying, in great detail, the degree of accuracy to which computations must be carried out, rounding behavior, error and exception handling, and the results of the basic floating-point arithmetic operations, comparisons, and conversions. The Standard also specified binary formats for floating-point numbers, formats that were rapidly adopted by the industry and are now widely supported in hardware and software.

The two most important floating-point data formats described in the IEEE 754 Standard are shown in Figure 2. The single-precision format occupies 32 bits (a double word for Intel processors). The

double-precision format requires 64 bits (a quad word for Intel CPUs). Both formats consist of three fields: a sign bit, which is always the most significant bit, followed by the binary exponent, with the mantissa in the remaining, least significant bits. Single-precision numbers can take on values in the (approximate) range $\pm 1.18*10^{-38}$ through $\pm 3.40*10^{38}$; double-precision numbers lie in the range $\pm 2.23*10^{-308}$ to $\pm 1.80*10^{308}$.

The sign bit is 1 if the number is negative and 0 if the number is positive. The mantissa is unsigned and does not change with the sign of the floating-point number. Because the mantissa is left-normalized, its most significant bit is (by definition) always 1. Consequently, the IEEE 754 designers pulled off a neat trick: they specified that the mantissa has an "implied leading bit," which is always 1 and is not present in the actual data. This allows an extra bit of precision to be squeezed out of each floating-point format.

The 8087 provided a hardware implementation of the entire IEEE 754 Standard.

The exponent field in both types of floating-point numbers is "biased"; that is, offset from zero by a fixed amount. For single-precision numbers, which have an 8-bit exponent, 127 (7Fh) in the exponent field corresponds to a true exponent value of 0. Double-precision numbers, which have an 11-bit exponent field, use an exponent bias of 1,023 (3FFh). This bias allows the reciprocal of any normalized floating-point number to be represented without underflow. The relative sizes of the exponent fields in the two formats were chosen so that they would allow a double-precision number to accommodate the product of as many as eight single-precision numbers without the possibility of overflow.

The exponent of an IEEE 754 floating-point number can also take on two "magic" values, causing the number to be handled as a special case. If all bits of the exponent are zero, then the floating-point number is either zero or a "denormalized" number—the result of a "graceful underflow" (more about this in a later installment). If all bits of the exponent are set, then the floating-point number represents either infinity or a special signalling value: NaN (Not a Number).

A couple of practical examples of binary floating-point data will help clarify the way the standard works. Consider the 4-byte (32-bit), single-precision floating-point number

```
41h 20h 00h 00h
```

We see that the sign bit is 0, the biased exponent is 10000010B or 82h, and the mantissa (after restoring the "implied leading bit") is

```
101000000000000000000000B
```

or A00000h. Correcting for the exponent bias, we have $1.010B*2^3$, or 10 decimal.

As another example, consider the double-precision floating-point number

```
BFh E0h 00h 00h 00h 00h 00h 00h
```

which occupies 8 bytes (64 bits). The sign bit is 1, the biased exponent is 01111111110B or 3FEh, and the mantissa (after inserting the "implied leading bit") is 10000000000000h. Correcting for the exponent bias, we have $-1.0B*2^{-1}$, or -0.5 decimal.

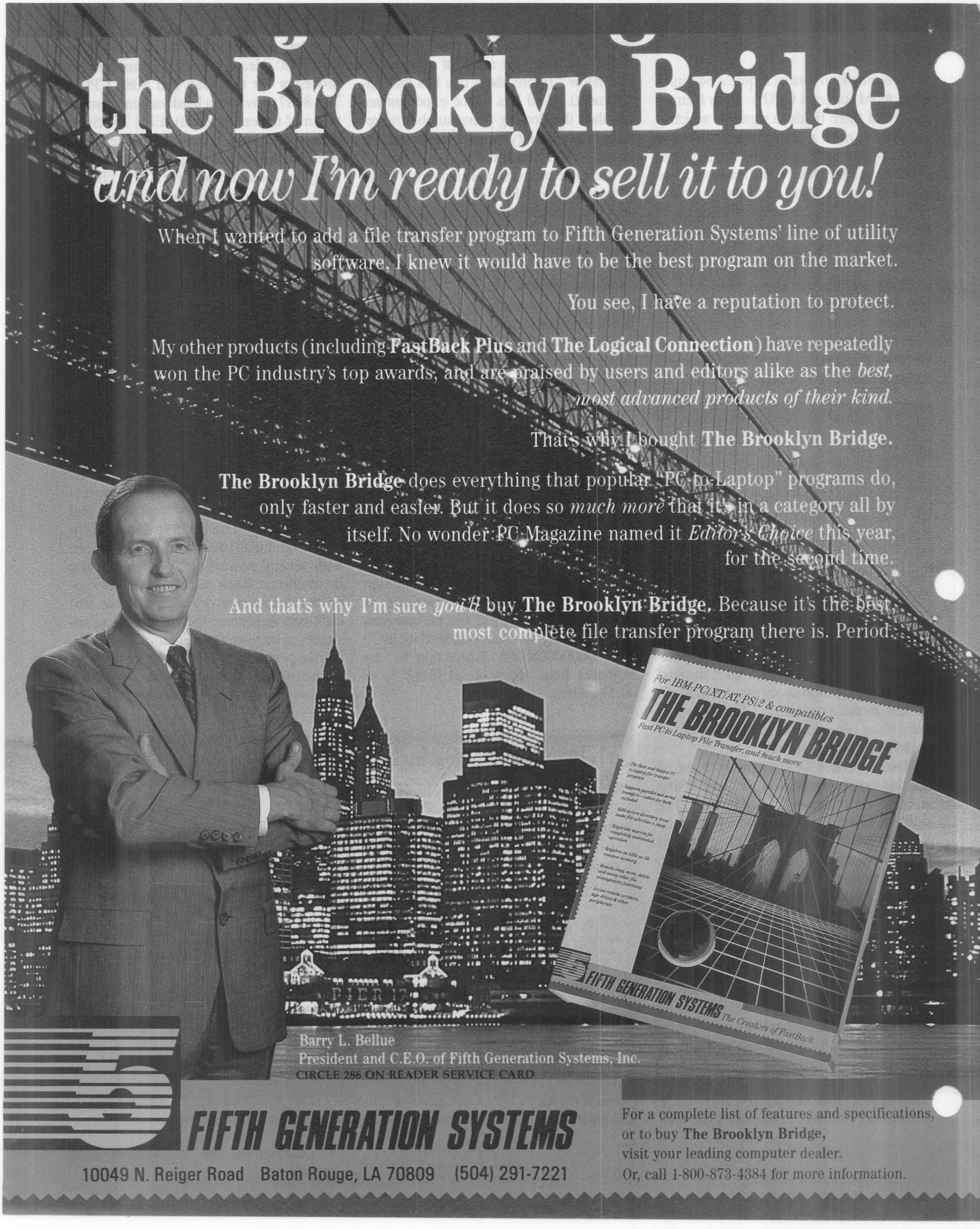
been less influential had it not been for Intel's 1980 release of the 8087 numeric coprocessor for the 8086 and 8088 CPUs. The 8087 provided a hardware implementation of the entire (draft) IEEE 754 Standard, even down to its most esoteric aspects (such as supporting two flavors of infinity: affine and projective). This chip itself quickly became the yardstick by which the compliance with the impending Standard of all other CPUs, numeric coprocessors, and software floating-point libraries was judged. The 8087 also brought an unprecedented (and largely unanticipated) amount of number-crunching power within the reach of every microcomputer user. This made it possible to migrate many demanding minicomputer and mainframe applications onto personal computers for the first time.

The 8087 was followed by the 80287 numeric coprocessor in 1983, and by the 80387 numeric coprocessor in 1987. Designed to work with the 80286 CPU, the 80287 was the first of the Intel coprocessors to support memory protection and multitasking. The 80387, designed to work with the 80386 CPU, was enhanced with several powerful new trigonometric instructions. Each successive chip supported a larger memory address space, and each benefited from the technological advances in large-scale integration made during the period. These included both an increase in clock speeds and a decrease in the number of machine cycles required for

SPECIAL IEEE-STANDARD EXPONENT VALUES

Exponent bits	Mantissa bits	Special meaning
all zero	all zero	floating-point zero
all zero	nonzero	denormalized floating-point number (usually result of "graceful underflow")
all set	all zero	infinity
all set	nonzero	"Not a Number" or "NaN" (various reserved mantissa values are used to signal overflow, unrecoverable underflow, invalid operands, invalid result, inexact result, and so on)

Figure 3: The IEEE 754 Standard reserves certain exponent values. Floating-point numbers with all bits zero or all bits set in the exponent field are trapped and receive special treatment.



the Brooklyn Bridge
and now I'm ready to sell it to you!
When I wanted to add a file transfer program to Fifth Generation Systems' line of utility software, I knew it would have to be the best program on the market.
You see, I have a reputation to protect.
My other products (including **FastBack Plus** and **The Logical Connection**) have repeatedly won the PC industry's top awards, and are praised by users and editors alike as the *best, most advanced products of their kind.*
That's why I bought **The Brooklyn Bridge.**
The Brooklyn Bridge does everything that popular "PC-to-Laptop" programs do, only faster and easier. But it does so *much more* that it's in a category all by itself. No wonder PC Magazine named it *Editor's Choice* this year, for the second time.
And that's why I'm sure *you'll* buy **The Brooklyn Bridge.** Because it's the best, most complete file transfer program there is. Period.
For IBM-PC/XT/AT, PS/2 & compatibles
THE BROOKLYN BRIDGE
Fast PC to Laptop File Transfer, and much more
FIFTH GENERATION SYSTEMS The Creators of FastBack
Barry L. Bellue
President and C.E.O. of Fifth Generation Systems, Inc.
CIRCLE 286 ON READER SERVICE CARD
FIFTH GENERATION SYSTEMS
10049 N. Reiger Road Baton Rouge, LA 70809 (504) 291-7221
For a complete list of features and specifications, or to buy **The Brooklyn Bridge**, visit your leading computer dealer.
Or, call 1-800-873-4384 for more information.

each floating-point operation. For a typical floating-point instruction mix, the performance of the 20-MHz 80386/387 combination is about 16 times better than that of a 5-MHz 8086/8087 duo.

The 8087, 80287, and 80387 chips are called "coprocessors" because they are closely coupled to the system's CPU, have a highly specialized instruction set, and cannot function alone. By sharing the same data and address bus as the main CPU, the coprocessors monitor the CPU's instruction stream as it is fetched from memory and flows by on the data bus. Floating-point instructions begin with a special "escape code" that is recognized and acted upon by the numeric coprocessor; the CPU essentially ignores the floating-point instructions except to perform address calculations on behalf of the numeric coprocessor when they are needed.

The 80*x*87 coprocessors were not the first floating-point arithmetic chips available for use with microprocessors. A number of early 8080, Z-80, and even 8086/88-based microcomputers had sockets for the AMD 9511 and 9512 chips, which supported 32-bit and 64-bit floating-point operations, respectively. But the AMD products were not coprocessors (they were addressed through an 8-bit I/O port like a peripheral device). Moreover, they used nonstandard data formats, were slow and clumsy to program, and enjoyed little if no support in commercial, mass-market software packages. The 80*x*87 chips were the first hardware number-crunchers that were cheap enough, and pervasive enough (thanks to the 8087 socket built into the very first PC motherboard), to motivate mass-market software publishers to have their programs check for the presence of a numeric coprocessor and use the coprocessor if it was available.

In the next installment, I'll discuss the architecture of the 80*x*87 series in more detail, and present routines that allow you to detect and exploit numeric coprocessors in your own programs.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan ■

Why settle for a single player,

by
Ray Duncan

Power Programming

Arithmetic Routines for your Computer Programs, Part 5

■ **Coprocessors first gave PCs the mathematical muscle of minis. Here's a look at how they work and how you can work them into your own programs.**

One of the most forward-looking features of the original IBM PC was an undocumented, empty 40-pin socket located very close to the 8088 CPU on the motherboard. Despite IBM's lack of official comment, it didn't take long for people who looked at the machine's schematic to decide that this socket just *must* have been designed for Intel's new 8087 numeric coprocessor. In due course, some brave soul plugged an 8087 into the mystery socket, turned on the juice, and breathed a sigh of relief when nothing in his precious $5,000 64K PC went up in smoke as a result.

Word spread rapidly through the still-minuscule community of PC users that the IBM PC was 8087-capable, but a couple of years passed before large numbers of people started installing 8087s. When the first PCs were shipped in late 1981, 8087s were still very scarce (many of the available chips were buggy, temperature-sensitive engineering samples), cost over $400 each, and were regarded as difficult to program. Indeed, 8087s were considered so esoteric that a company called MicroWay built a successful business on buying the coprocessors in quantity from Intel and reselling them, with installation instructions, to anxious end-users who also valued the company's friendly telephone support, test programs, and hand-coded libraries for a few popular compilers.

The combination of the PC and the 8087 brought unprecedented number-crunching power to desktop computers. By itself, the IBM PC was barely faster at floating-point calculations than an Apple II or a Z-80 machine, but the addition of an 8087 made the PC powerful enough to handle many applications that had formerly required mainframes or minicomputers. The 8087 implemented the ANSI/IEEE 754 1981 Draft Standard for Binary Floating Point Arithmetic in a single integrated circuit. The standardized architecture of the PC/8087 combination and the potentially enormous user base also provided an attractive target market for microcomputer software developers, who had shown little interest in previous floating-point chips, such as the AMD 9511 and 9512.

The original 8087 worked with either the 8086 or 8088 processors and was first shipped in 1980. The 80287 was designed to work with the 80286 CPU and was the first of the Intel coprocessors to support memory protection and multitasking; it was shipped in 1983. The 80387 was designed to work with the 80386 CPU and was enhanced with several powerful new trigonometric instructions. After being brought into conformity with the ANSI/IEEE 754 Standard (as it was finally approved in 1985), the 80387 arrived on the scene in 1987. The 80387 is the last of its line—the 80486 has all of the logic of the 80387 built in and does not require a separate numeric coprocessor at all. The relative performance of the various 80*x*86/80*x*87 combinations is summarized in Figure 1.

(Note: Throughout the remainder of this column, I'll use "CPU" to refer to any member of the Intel 80*x*86 processor family (8086, 8088, 80286, 80386, or 80486), and "coprocessor" to refer to any member of the Intel 80*x*87 family (8087, 80287, or 80387). Statements intended to apply to a specific member of either product family will include an explicit reference to a particular model number.)

80x87 ARCHITECTURE

The 8087, 80287, and 80387 chips are called *coprocessors* because they are

PC MAGAZINE INTEL COPROCESSOR HISTORY

Processor/ coprocessor	Typical speed	First shipped	Relative performance
8086 and 8087	5 MHz	1980	1
80286 and 80287	8 MHz	1983	2.5
80386 and 80387	20 MHz	1987	16
80486 with embedded 80387	25 MHz	1989	64

Figure 1: A comparison of the speeds, dates first available, and relative performance of the various combinations of Intel CPUs and their coprocessors.

closely coupled to the system's CPU, have a highly specialized instruction set, and cannot function alone. These coprocessors share the same data bus and address bus as the main CPU. They also monitor the CPU's instruction stream as it is fetched from memory and flows by on the data bus. Floating-point instructions all begin with a special escape code that is recognized and acted upon by the coprocessor. The CPU essentially ignores the floating-point instructions except to perform any needed address calculations on behalf of the coprocessor.

The shared instruction stream and the fact that many of the more complex coprocessor instructions can require hundreds of cycles to complete let careful programmers achieve some concurrent processing.

The shared instruction stream and the fact that many of the more complex coprocessor instructions can require hundreds of cycles to complete allow the careful programmer to achieve a certain amount of *concurrent processing*. A program can load a floating-point number onto the coprocessor, issue a floating-point square root instruction, and then execute several dozen CPU instructions in the 100+ cycles required for the square root operation to complete. The program can then resynchronize with the coprocessor and either unload the result of the square root operation into memory or use that result as an argument for the next floating-point operation.

Obviously, such concurrent program-

INSTRUCTION CATEGORIES FOR 80X87 COPROCESSORS

DATA TRANSFER	
FLD, FST, FSTP	Load or store floating-point value
FILD, FIST, FISTP	Load or store integer value
FBLD, FBSTP	Load or store binary-coded-decimal (BCD) value
FXCH	Exchange two coprocessor registers
ARITHMETIC	
FADD, FADDP, FIADDP	Floating-point add
FSUB, FSUBP, FISUB	Floating-point subtract
FSUBR, FSUBRP, FISUBR	Floating-point subtract reversed
FMUL, FMULP, FIMUL	Floating-point multiply
FDIV, FDIVP, FIDIV	Floating-point divide
FDIVR, FDIVRP, FIDIVR	Floating-point divide reversed
FSQRT	Floating-point square root
FSCALE	Scale one value by another
FPREM, FPREM1	Partial remainder (FPREM1 on 80387 only)
FRNDINT	Round to integer
FXTRACT	Extract exponent and mantissa from value
FABS	Absolute value
FCHS	Change sign of value
COMPARISON	
FCOM, FCOMP, FCOMPP	Compare two ordered values
FUCOM, FUCOMP, FUCOMPP	Compare two unordered values (80387 only)
FTST	Compare value to zero, set flags
FXAM	Test type of value, set flags
TRANSCENDENTAL	
FPTAN, FPATAN	Tangent and arctangent (partial on 8087 and 80287; generalized on 80387)
FSIN, FCOS, FSINCOS	Sine and cosine (80387 only)
F2XM1	Raise 2 to power and subtract one
FYL2X	Multiply value times $\log_2$ of another value
FYL2XP1	Multiply value times $\log_2$ of another value plus one
CONSTANTS	
FLDZ	Load the value zero
FLD1	Load the value one
FLDPI	Load the value pi
FLDL2T	Load the value $\log_2(10)$
FLDL2E	Load the value $\log_2(e)$
FLDLG2	Load the value $\log_{10}(2)$
FLDLN2	Load the value $\log_e(2)$
PROCESSOR CONTROL	
FINIT	Initialize coprocessor
FSTSW	Store status word (FSTSW AX form not available on 8087)
FLDCW, FSTCW	Load or store control word
FCLEX	Clear coprocessor exception (error) flags
FLDENV, FSTENV	Load or store coprocessor environment
FSAVE, FRSTOR	Save or restore coprocessor state
FINCSTP, FDECSTP	Increment or decrement coprocessor stack pointer
FFREE	Mark floationg-point register as empty
FNOP	Coprocessor "no-operation"
FENI, FDISI	Enable or disable coprocessor error interrupts (8087 only; ignored on 80287 and 80387)

Figure 2: A summary of the instruction set of the 8087, 80287, and 80387 numeric coprocessors.

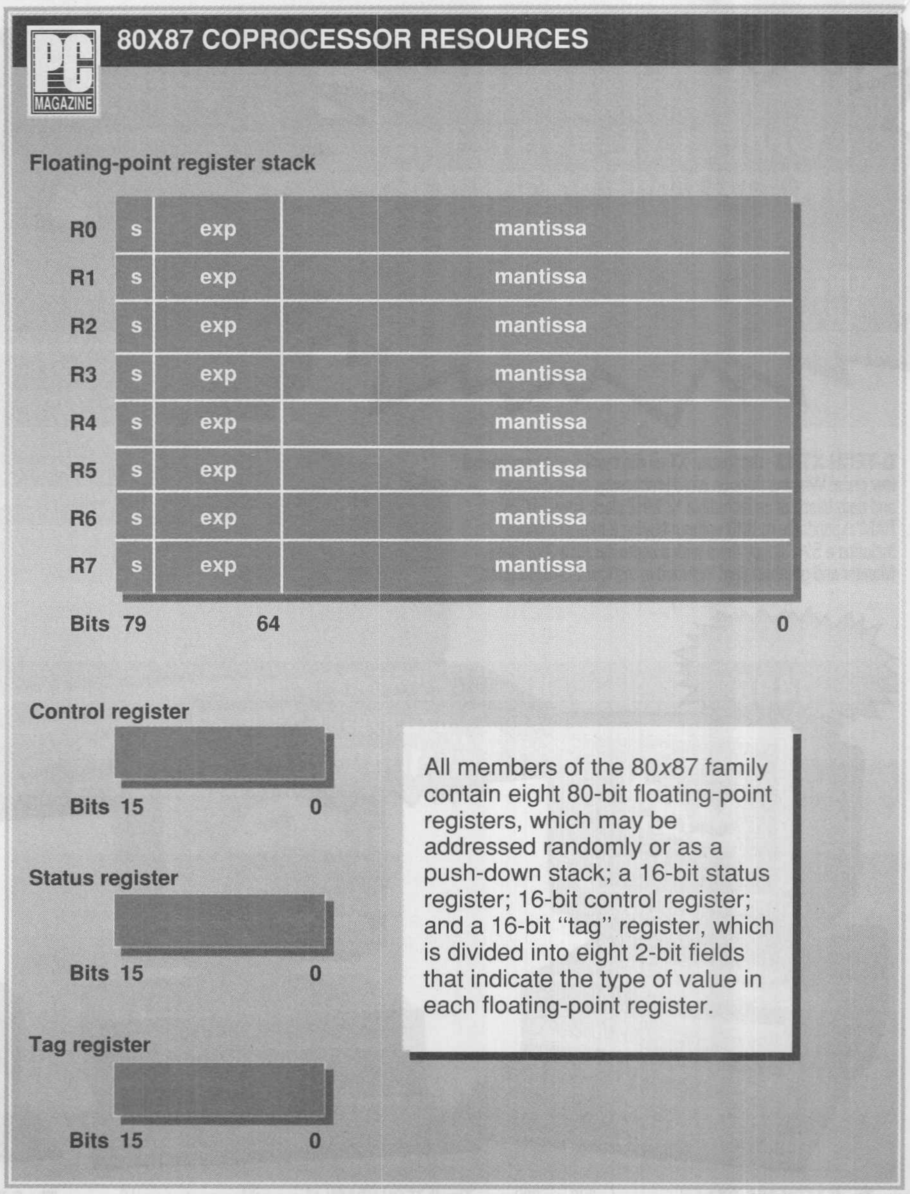
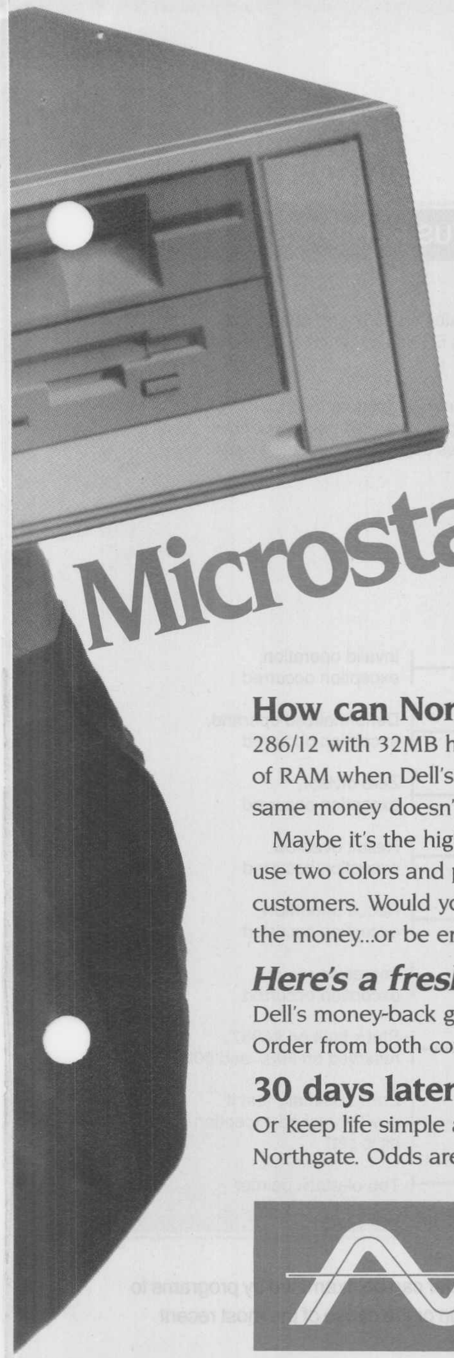


Figure 3: Special coprocessor instruction pointer and data pointer registers are also common resources of 80*x*87 chips. They're located on the CPU and used by floating-point error handlers.

ming requires careful scheduling of operations and counting of cycles so that both processors will be kept as busy as possible and will not waste time waiting on each other. The required coding is also extremely environment-dependent, since instruction timings vary widely among the different models of CPUs and numeric coprocessors.

The instructions understood by 80*x*87 coprocessors fall into six basic categories: data transfer, arithmetic, comparison, transcendental, constants, and processor control. These are summarized in Figure 2. Mnemonics ending with "P" automatically pop one operand off the coprocessor's floating-point register stack; mnemonics ending with "PP" pop two operands. The instruction sets of the 80287 and 80387 are proper supersets of the 8087; the 80287 and 80387 will run 8087 application code without modification. However, considerably better performance and more powerful programs can be obtained by modifying the source code to take advantage of the improved synchronization characteristics and instruction sets of the 80287 and 80387. The generalized



Microstation with...

How can Northgate afford to offer a 286/12 with 32MB hard drive and one full megabyte of RAM when Dell's newest Series 210 system, for the same money doesn't match up?

Maybe it's the high cost of color advertising. We use two colors and pass the savings on to our customers. Would you rather get more computer for the money...or be entertained with color pictures?

Here's a fresh idea... Make Northgate and Dell's money-back guarantee program meaningful: Order from both companies.

30 days later send back the loser.
Or keep life simple and place your order with Northgate. Odds are it's the one you'll keep anyway.

NORTHGATE COMPUTER SYSTEMS™

"We hear you!" **Northgate Computer Systems, Inc.**
P.O. Box 41000,
Plymouth, Minnesota 55441

CALL TOLL-FREE 24 HOURS EVERY DAY

800-548-1993

Canada: 800-338-8383

FINANCING: Use the Northgate Big 'N' revolving credit card. We have millions in financing available. We accept your Visa or MasterCard too. Lease it with Northgate, up to five-year terms available.

Prices and specifications are subject to change without notice. Northgate reserves the right to substitute components of equal or greater quality or performance. All items subject to availability.

P.S. Ask about OmniKey/PLUS, the new Northgate Keyboard designed the way you want!

1 Full Megabyte RAM
100% GREATER!

32 Mbyte Hard Drive
50% LARGER!

12 MHz Processor Speed

$1699.00

BUYER'S SCORE SHEET.
Read the Specs. Check the Leader.
Total the winning checks and make your decision.

Standard Features	DELL	✓	NORTHGATE	✓
Processor	12.5 MHz 80286		12 MHz 80286	
Memory	512K		One Megabyte	
Video Interface	16 Bit Built-In (Factory Fixable) Only		16 Bit Add-on (On-Site Fixable)	
Display	12" VGA Mono .31 DP		12" VGA Mono .31 DP	
Floppy Drive	One - 1.2 or 1.44		One - 1.2 or 1.44	
Std. Hard Drive Capacity	20 Mbyte		32 Mbyte RLL	
Hard Drive Type	IDE Built-In		Can use - RLL, MFM, IDE, ESDI	
I/O Capabilities	2 Ser., 1 PP		2 Ser., 1 PP	
Keyboard	"Mushy Touch" 101		Famous OmniKey/102	
Software	Diagnostic		On-Line Help, MS-DOS	
Space Saver Case	15"W, 15"L, 4"H		14.5"W, 16.5"L, 5.5"H	
Moneyback Period	30 Days		30 Days	
Warranty	1 Yr. Parts & Labor		1 Yr. Parts & Labor	
Phone Tech Support	Unlimited, Toll Free		Unlimited, Toll Free	
Hours Open - Sales	Standard Daytime, Eve		24 Hours All Day Every Day	
Hours Open - Tech	Standard Daytime		24 Hours All Day Every Day	
Total	$1,699		$1,699	
SCORE	DELL		NORTHGATE	

And the winner is...

Call NOW!

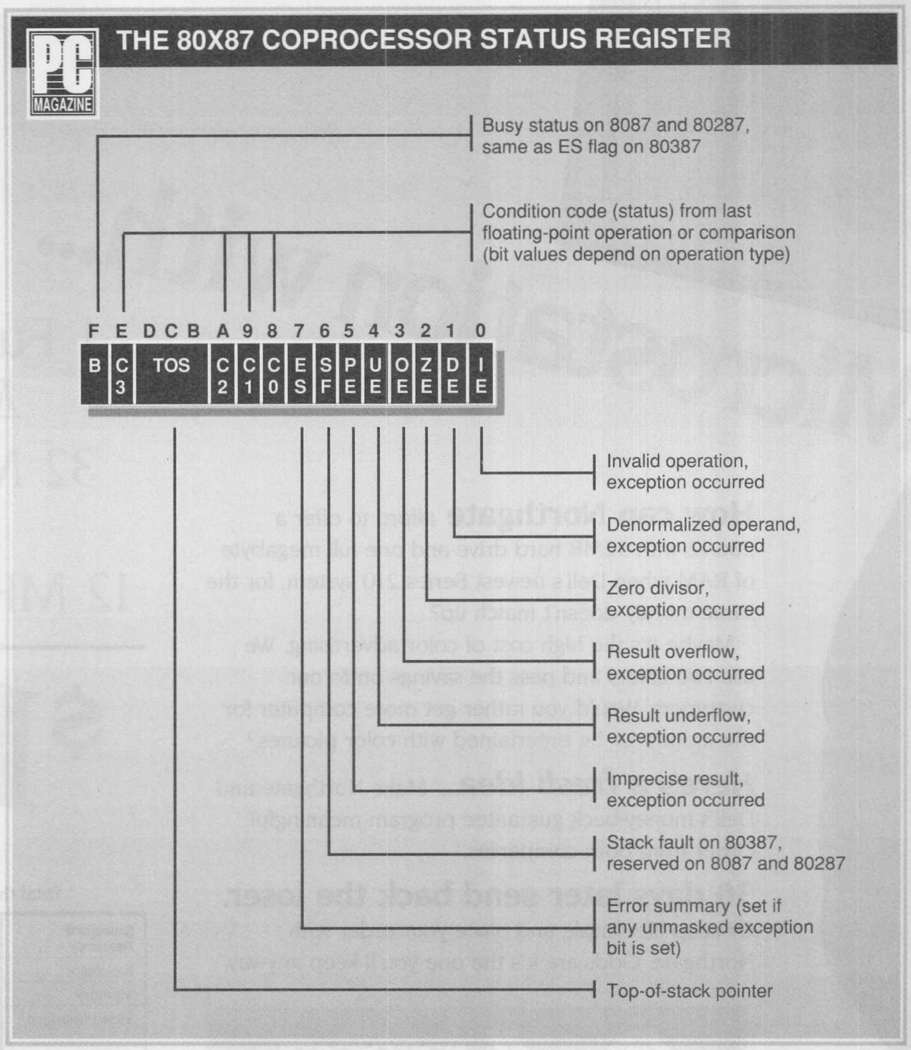


Figure 4: The read-only word in the coprocessor status register can be examined by programs to determine the result of the most recent floating-point operation or the cause of the most recent floating-point exception (error).

tangent, arctangent, sine, and cosine instructions, which are unique to the 80387, are particularly important in these respects.

As illustrated in Figure 3, the basic on-chip resources that are common to all members of the 80*x*87 family are eight floating-point registers, a tag word, a control word, and a status word. The floating-point registers are each 80 bits wide and are generally used as a push-down stack. (Each register can instead be addressed directly by the programmer when necessary.) The tag word is divided into eight 2-bit fields, each of which corresponds to a floating-point register. Each of these fields indicates whether the register is empty or whether the number in that register is valid or invalid, a floating-point value, zero, or infinity.

The status word, which is read-only, contains the condition codes for the most recent floating-point operation, error indicators, and also a specification of which floating-point register is the current top of stack. This is diagrammed in Figure 4. If you push too many numbers onto the 80*x*87 stack and the top-of-stack pointer wraps, an exception is generated. Similarly diagrammed in Figure 5, the control word is read/write and determines the rounding mode of the chip and the precision to which operations are carried out. "Mask" bits in the control word determine which types of errors are allowed to generate an interrupt.

THE 80X87 COPROCESSOR CONTROL REGISTER

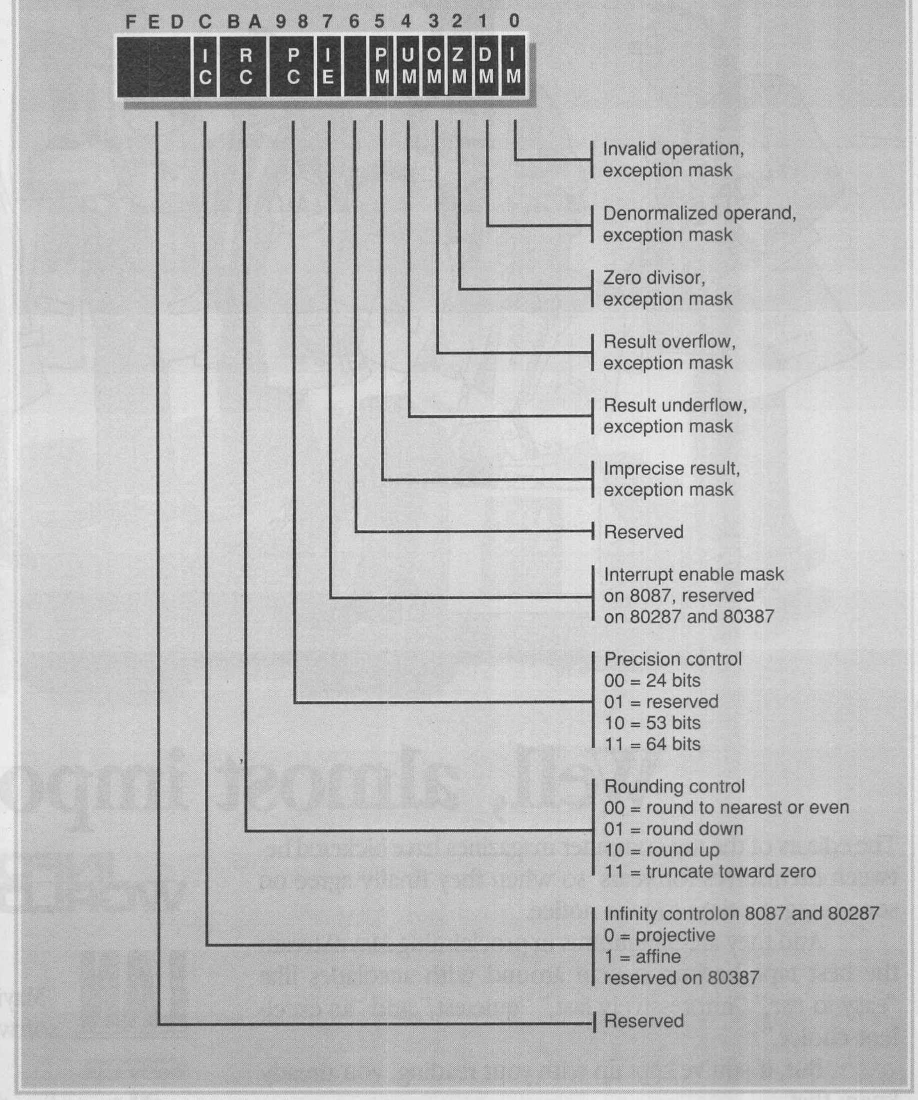


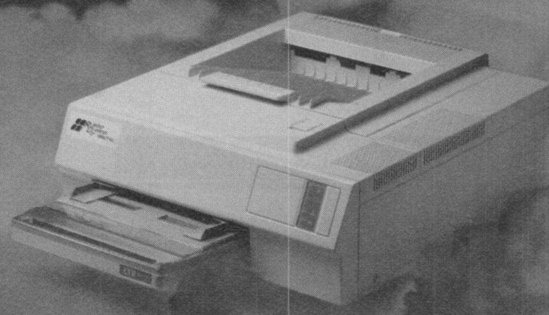
Figure 5: The word in the control register can be read or written by application programs to query or set the current rounding mode and precision mode and to control error handling. When the exception mask bit for a particular error type is set to 1, that error type will not generate an interrupt, and the value of the result will be set to a special bit pattern called a NaN (Not-a-Number).

In addition to the registers that are on board the coprocessor, there are special coprocessor instruction pointer and data pointer registers located on the CPU chip that are associated with the coprocessor's execution of floating-point instructions. When a coprocessor error interrupt occurs, the interrupt handler can examine these registers to determine the location and type of the floating-point instruction and/or operand that caused the problem. (Since the CPU and coprocessor generally run asynchronously, the values in the CPU's own instruction pointer and registers are not likely to be helpful.)

80x87 DATA FORMATS

Intel CPUs and coprocessors communicate primarily by means of shared memory. Thus, for example, if you want to use a certain number in a calculation and it happens to be in a CPU register, the only way

A Type Shop's NIGHTMARE...



A Designer's Dream!

1000 DPI Plain-Paper Typesetting

Desktop publishing is not only here, it's thriving. Every day, more graphic designers are learning to use computers as tools. However, at the standard resolution of 300 x 300 dpi, most laser-printed output is used primarily for proofing copy. Final copy is still sent to the typesetter.

Like a dream come true, Raster Devices Direct, Inc.™ presents an alternative to type shops: the Trendsetter 1000™.

The Trendsetter 1000 is an IBM PC/XT/AT or '386 compatible laser printer that provides 1000 x 400 dpi Plain-Paper Typesetting output. It comes standard with 35 Bitstream® Fontware™ scalable fonts and is compatible with over 200 more. Plus, the Trendsetter 1000 offers two different printing modes: a Direct mode, and a PostScript® compatibility mode. Use the Direct mode for super-fast 1000 x 400 dpi printing from any GEM, Windows, or other supported application. (You can also use the Direct mode to create font special effects like shading, patterns, squeezing, stretching, shearing, obliquing and rotating.) Use the PostScript compatibility mode to print 1000 x 400 dpi output from PostScript language files. The Trendsetter 1000 provides Plain-Paper Typesetting for *all* your publishing needs.

So if you use graphics programs, desktop publishing, CAD/HPGL, or even advanced word processors, and have dreamt of 1000 dpi Plain-Paper Typesetting for just **$5,995** call Raster Devices Direct at 1-800-468-1732. (In Minnesota, call 612-941-4919.) Send us your Corel DRAW, Micrografx Designer, GEM Artline, Aldus PageMaker, Xerox Ventura Publisher, AutoCAD, Microsoft Word 5.0, WordPerfect 5.0 , or HPGL files and we'll send you back 1000 dpi Plain-Paper Typesetting output!

CIRCLE 192 ON READER SERVICE CARD

Raster Devices Direct, Inc.™

PC Publishing Power Direct to You

1-800-468-1732 FAX (612) 941-5116

VISA, MASTERCARD, and AMERICAN EXPRESS

All type is actual output produced from the Trendsetter 1000™

tion there is no direct register-to-register communication between the CPU and the coprocessor. (The exception is that, on the 80287 and 80387, the FSTSW instruction is able to write the coprocessor's status word directly into the CPU's AX register. This expedites testing the results of a floating-point operation for conditional branching.)

The coprocessor family supports 7 different data types in memory: 16-, 32-, and 64-bit integers; 80-bit packed binary coded decimal (BCD); and 32-, 64-, and 80-bit floating-point numbers. These are shown in Figure 6. The 32-bit and 64-bit floating-point formats are identical to those defined by the ANSI/IEEE 754 Standard. The 16- and 32-bit integer formats are identical to those supported by the entire 80*x*86 family. And the 64-bit integer format is the same as the double-precision integers used by the 80386 machine when it is running in 32-bit protected mode. Regardless of a number's format in memory, however, when it is loaded into a coprocessor floating-point register it is always converted to the 80-bit "extended-precision" floating-point format (which is also called the "temporary real" data type in some Intel manuals).

The use of the extended-precision floating-point format for all operations that are internal to the 80*x*87 coprocessor family has some interesting consequences. First,

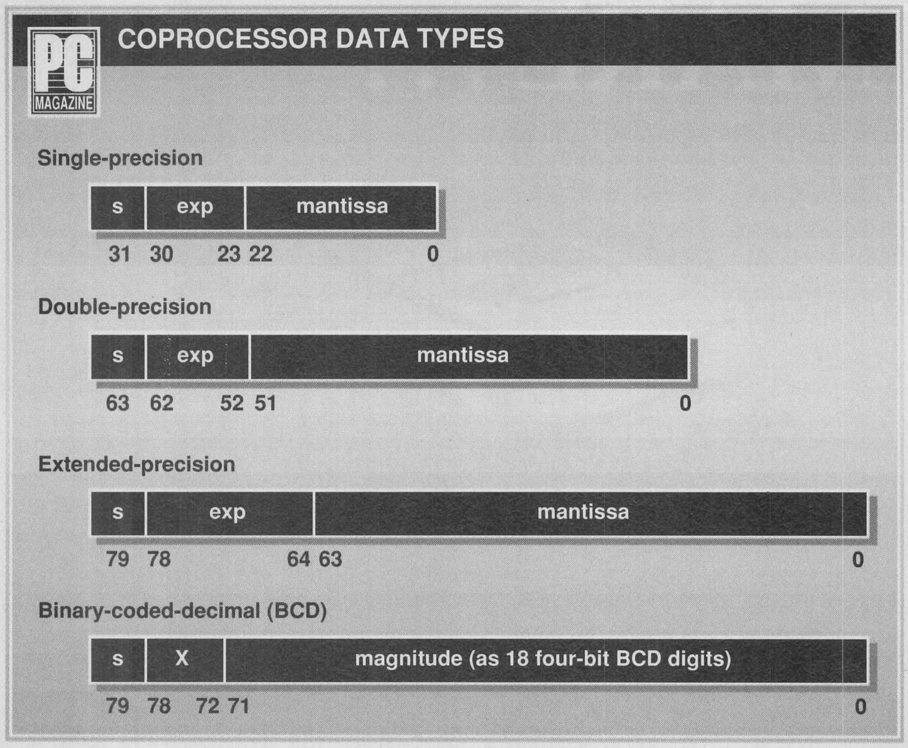


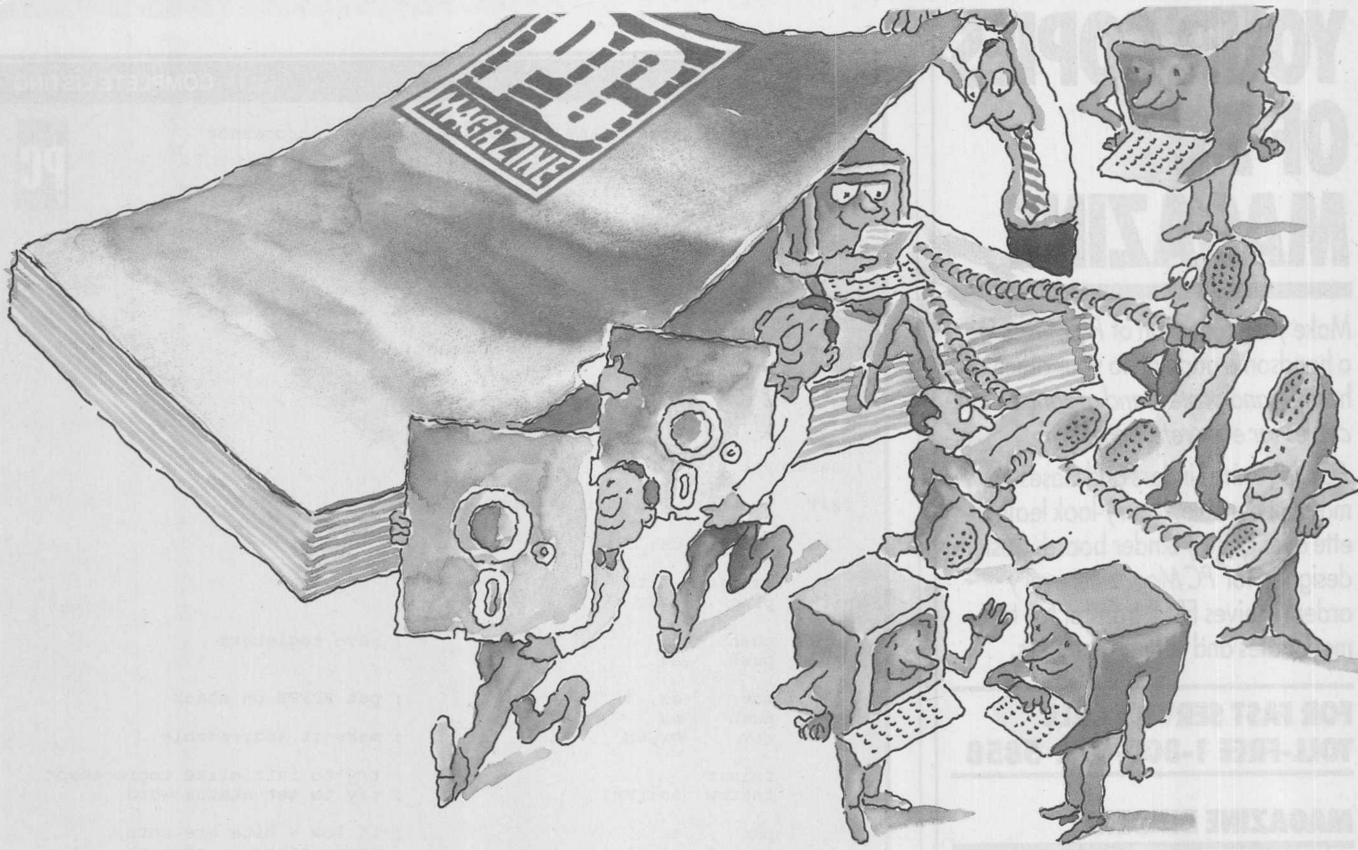
Figure 6: Floating-point and binary-coded-decimal data formats supported by the Intel 80*x*87 numeric coprocessor family in memory. The single-precision and double-precision floating-point formats are as specified in the ANSI/IEEE 754 standard. The binary-coded-decimal (BCD) format packs two digits per byte, giving a total of 18 digits plus a sign bit in an 80-bit format (*x* represents 7 bits that are unused). The 16-, 32-, and 64-bit integers (not shown here) are normal two's complement binary integers with the sign in the most significant bit. Regardless of a number's format in memory, when it is loaded into a floating-point register on the coprocessor, it is always converted into the extended (80-bit) floating-point format.

SPECIAL IEEE-STANDARD EXPONENT VALUES

Exponent bits	Mantissa bits	Special meaning
all zero	all zero	floating-point zero
all zero	nonzero	denormalized floating-point number (usually result of "graceful underflow")
all set	all zero	infinity
all set	nonzero	"Not a Number" or "NaN" (various reserved mantissa values are used to signal overflow, unrecoverable underflow, invalid operands, invalid result, inexact result, and so on)

Figure 7: Floating-point numbers in which all bits are zero or all bits are set in the exponent field are trapped and receive special treatment in accordance with the ANSI IEEE 754 binary floating-point standard.

Intel CPUs and coprocessors communicate primarily via shared memory rather than direct register-to-register. So to use a number in a CPU register in a calculation you must store it in RAM first.



Get the Most Out of Us!

Join PC MagNet and have instant access to PC MAGAZINE's programs and product reviews.

PC MagNet is *PC MAGAZINE'S* on-line information service, which is administered by CompuServe. A PC MagNet account gives you:

- Access to all of *PC MAGAZINE'S* utilities and programs
- The latest PC Labs Benchmark tests
- New versions of *PC MAGAZINE'S* utilities
- A way to talk directly with *PC MAGAZINE* editors and columnists
- On-line support for our programs in the forums
- Direct entry to the Computer Library family of databases
- A downloadable-index to *PC MAGAZINE'S* product reviews

One call gets you started!

Sign up on-line at no additional charge.

1. To subscribe, set your communications software for either 300, 1,200 or 2,400 bps, 7 data bits, even parity, 1 stop bit, and full duplex. To find the number nearest your dialing exchange, dial (800) FIND CIS (346-3247). When the modem connects, press <Enter>. At the HOST NAME prompt, enter PHONES. Follow the menus and note the number closest to you. After you have found the phone number, hang up and dial using the number you just found.
2. When you receive a CONNECT or CARRIER DETECTED message, enter **Ctrl-C.**
3. At the HOST NAME prompt, enter **CIS.**
4. At the USER ID prompt, enter **177000,5000.**
5. At the PASSWORD prompt, enter **PC*MAGNET.**
6. At the Enter agreement number prompt, enter **Z10D9001.**
7. Register your name and enter your **VISA, MasterCard,** or **American Express** account number. You may choose to have your company billed directly by choosing Business Account Invoicing. For more information about Business Account Invoicing, call (800) 848-8199.
8. Your personal **User ID** number and **Password** will be displayed at the end of the subscription process. Please record them in a secure place.
9. A new password will arrive in the mail within 10 days to confirm your subscription.

There are no connect charges for PC MagNet during the subscription process. PC MagNet costs $12.80 an hour for 1,200/2,400-bps service and $6.30 for 300-bps, via MasterCard, VISA, or American Express. For assistance regarding passwords, membership, billing or payment terms, call Customer Assistance at (800) 848-8990; outside the United States, call (614) 457-8650. There are no upfront subscription fees, or monthly charges. You pay only for the time spent on-line.

CompuServe members can join by entering GO PCMAG at any CompuServe prompt.

PC MAGNET

CIRCLE 497 ON READER SERVICE CARD

INIT87.ASM **COMPLETE LISTING**

```
        title   INIT87.ASM Initialize Numeric Coprocessor
        page    55,132

; INIT87.ASM    Initialize 80x87 Numeric Coprocessor
;
; Copyright (C) 1989 Ziff Davis Communications
; PC Magazine * Ray Duncan
;
; Call with:    AX      = control word for desired rounding
;                         mode, precision, exception mask
;
; Returns:      (if coprocessor present)
;               Z flag  = True (1)
;
;               (if coprocessor not found)
;               Z flag  = False (0)
;
; Destroys:     nothing

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT

        public  init87
init87  proc    near

        push    bx                      ; save registers
        push    ax

        mov     ax,-1                   ; put FFFFH on stack
        push    ax
        mov     bx,sp                   ; make it addressable

        fninit                          ; try to initialize coprocessor
        fnstsw  ss:[bx]                 ; try to get status word

        pop     ax                      ; if low 8 bits are zero,
        or      al,al                   ; coprocessor is present

        jnz     initx                   ; jump if no coprocessor

        fldcw   ss:[bx+2]               ; load coprocessor control word

initx:  pop     ax                      ; restore registers
        pop     bx
        ret                             ; and return result in Z flag

init87  endp

_TEXT   ends

        end
```

Figure 8: This routine tests for the presence of an 80x87 numeric coprocessor and, if one is present, configures it for the desired rounding, precision, and error-handling modes.

since the dynamic range of the 80-bit format is so much greater than that of the 32-bit or 64-bit floating-point formats typically used in program variables, an overflow or underflow of a final result is quite rare, assuming that all intermediate results are maintained on the chip. Second, conversion of a number from one data type to another in memory is trivial: you just load the original data from memory onto the coprocessor, and then unload it again in the desired format.

Let's take a closer look at the coprocessor's extended floating-point data type, which corresponds to the optional double extended format of the ANSI/IEEE 754 Standard. (I discussed the single-precision and double-precision floating-point number formats here last time.) The dynamic range of these numbers is approximately $\pm 3.4*10^{-4932}$ to $\pm 1.2*10^{4932}$. The 80 bits are divided into three fields: a sign bit, a 15-bit exponent (sometimes called a "characteristic"), and a 64-bit mantissa (or "significand").

The sign bit is 1 if the number is negative and 0 if the number is positive. The mantissa is unsigned and does not change with the sign of the floating-point number. Because the mantissa is left-normalized, its most significant bit is always 1 unless the number is zero or unless it is the result

DIMUL87.ASM **COMPLETE LISTING**

```
        title   DIMUL87.ASM 80x87-based Signed Divide
        page    55,132

; DIMUL87.ASM   Double Precision Signed Integer Multiply
;               for 80x87 coprocessor and 8086, 8088, 80286, or
;               80386 in real mode/16-bit protected mode
;
;               Be sure to call INIT87 routine first to test for
;               coprocessor existence and to set rounding mode,
;               precision, and exception masks!
;
; Copyright (C) 1989 Ziff Davis Communications
; PC Magazine * Ray Duncan
;
; Call with:    DX:AX           = double-precision argument 1
;               CX:BX           = double-precision argument 2
;
; Returns:      DX:CX:BX:AX     = quad-precision product
;
; Destroys:     nothing

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT

        public  dimul
dimul   proc    near

        push    dx                      ; put argument 1 on stack
        push    ax

        push    cx                      ; put argument 2 on stack
        push    bx

        mov     bx,sp                   ; make arguments addressable

        fild    dword ptr ss:[bx]       ; load one argument
        fimul   dword ptr ss:[bx+4]     ; multiply it by the other

        fistp   qword ptr ss:[bx]       ; unload the result
        fwait                           ; wait for it to arrive

        pop     ax                      ; retrieve result
        pop     bx
        pop     cx
        pop     dx
        ret                             ; and exit

dimul   endp

_TEXT   ends

        end
```

Figure 9: This performs signed integer double-precision multiplication using the coprocessor.

of an operation that underflowed or had invalid operands. Unlike the single-precision and double-precision floating-point data formats, no implied leading bit is used in the mantissa of the extended-precision format.

The exponent is "biased"—offset from zero—by the value 16,383 (3FFFH). For example, a value of 16,386 in the exponent fields corresponds to an exponent of 3—in other words, the mantissa is multiplied by 2^3. Use of a biased exponent ensures that the reciprocal of any normalized floating-point number can be represented without underflow. The exponent can also take on two "magic" values that cause the number to be handled in a special way. If all bits of the exponent are zero, then the number is either zero or is a "denormalized" number—the result of a "graceful underflow." If all bits of the exponent are set, then the floating-point number represents either infinity or a special signalling value called a NaN (Not-a-Number). These combinations and their meanings are shown in Figure 7.

ELEMENTARY COPROCESSOR USAGE

Before an application program can make use of a coprocessor, it must first verify the coprocessor's presence in the host machine and then configure it for the desired

DIDIV87.ASM **COMPLETE LISTING**

```
        title   DIDIV87.ASM 80x87-based Signed Divide
        page    55,132

; DIDIV87.ASM   Double Precision Signed Integer Divide
;               for 80x87 coprocessor and 8086, 8088, 80286, or
;               80386 in real mode/16-bit protected mode
;
;               Be sure to call INIT87 routine first to test for
;               coprocessor existence and to set rounding mode,
;               precision, and exception masks!
;
; Copyright (C) 1989 Ziff Davis Communications
; PC Magazine * Ray Duncan
;
; Call with:    DX:CX:BX:AX     = quad-precision dividend
;               SI:DI           = double-precision divisor
;
; Returns:      DX:AX           = double-precision quotient
;               CX:BX           = double-precision remainder
;
; Destroys:     nothing

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT

        public  didiv
didiv   proc    near

        push    dx                      ; put dividend on stack
        push    cx
        push    bx
        push    ax

        push    si                      ; put divisor on stack
        push    di

        mov     bx,sp                   ; make arguments addressable

        fild    dword ptr ss:[bx]       ; put divisor on coprocessor
        fild    qword ptr ss:[bx+4]     ; put dividend on coprocessor

        fld     st(1)                   ; make copies of both
        fld     st(1)

        fdivrp  st(1),st(0)             ; perform signed divide

        fistp   dword ptr ss:[bx]       ; unload quotient

        fprem                           ; calculate remainder

        fistp   dword ptr ss:[bx+4]     ; unload remainder
        fstp    st(0)                   ; discard stack top

        pop     ax                      ; quotient into DX:AX
        pop     dx

        pop     bx                      ; remainder into CX:BX
        pop     cx

        add     sp,4                    ; clean up stack
        ret                             ; and exit

didiv   endp

_TEXT   ends

        end
```

Figure 10: This performs signed integer double-precision division with the numeric coprocessor.

rounding, precision, and error handling modes. One method of doing this is illustrated by the procedure INIT87.ASM, shown in Figure 8. The routine will first execute a no-wait form of the coprocessor initialization instruction FINIT, then it will attempt to transfer the status word from the coprocessor into a memory word that has been initialized to FFFFH with the FSTSW instruction. If the coprocessor is present, the procedure sets the desired modes by loading the coprocessor control word and then returns a status code in the CPU's zero flag.

The DIMUL87.ASM and DIDIV-87.ASM routines, which are listed in Figures 9 and 10, illustrate some basic principles of coprocessor programming. They use the coprocessor to implement double-precision integer (32-bit) signed multiplication and division, and are thus symmetric with the software-only unsigned multiply and divide routines DMUL.ASM and DDIV.ASM, published here in our November 14, 1989, issue. Note that the coprocessor does not directly support unsigned integer multiplies or divides, so I couldn't code direct equivalents for the previous DMUL.ASM and DDIV.ASM routines without going to a great deal of trouble to handle the upper half of the unsigned range.

> **Before an application program can make use of a coprocessor, it must first verify its presence in the host machine.**

The interactive demo programs TRY-DIMUL and TRYDIDIV illustrate the use of INIT87, DIMUL87, and DIDIV87. They test the status of the coprocessor, prompt you for two arguments, carry out the multiplication or division operation on the coprocessor, and then display the results. The source code for these two demo programs can be downloaded from PC MagNet.

In the next issue, we'll proceed to a discussion of coprocessor floating-point operations, proper use of the WAIT (FWAIT) instruction on the various coprocessor models, use of the various coprocessor rounding modes, and error handling.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:

PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan ■